PriorMSM: An Efficient Acceleration Architecture for Multi-Scalar Multiplication

CHANGXU LIU, State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China
HAO ZHOU, State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China
PATRICK DAI, Semisand Chip Design Pte. Ltd., Singapore, Singapore
LI SHANG, State Key Laboratory of Integrated Chips and Systems, School of Computer Science, Fudan University, Shanghai, China
FAN YANG, State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China

Multi-Scalar Multiplication (MSM) is a computationally intensive task that operates on elliptic curves based on $GF(P)$. It is commonly used in zero-knowledge proof (ZKP), where it accounts for a significant portion of the computation time required for proof generation. In this article, we present PriorMSM, an efficient acceleration architecture for MSM. We propose a Priority-Based Scheduling Mechanism (PBSM) based on a multi-FIFO and multi-bank architecture to accelerate the implementation of MSM. By increasing the pairing success rate of internal points, PBSM reduces the number of bubbles in the pipeline of point addition (PADD), consequently improving the data throughput of the pipeline. We also introduce an advanced parallel bucket aggregation algorithm, leveraging PADD's fully pipelined characteristics to significantly accelerate the implementation of bucket aggregation. We perform a sensitivity analysis on the crucial parameter of window size in MSM. The results indicate that the window size of the MSM significantly impacts its latency. Area-Time Product (ATP) metric is introduced to guide the selection of the optimal window size, balancing the performance and cost for practical applications of subsequent MSM implementations. PriorMSM is evaluated using the TSMC 28 nm process. It achieves a maximum speedup of 10.9× compared to the previous custom hardware implementations and a maximum speedup of 3.9× compared to the GPU implementations.

CCS Concepts: • **Hardware** → **Application-specific VLSI designs**; • **Computer systems organization** → *Architectures*; • **Security and privacy** → *Hardware security implementation;*

This work was supported in part by the National Key R&D Program of China under Grant 2023YFB2704600, and in part by the National Natural Science Foundation of China (NSFC) Research Projects under Grant 92373207 and Grant 62090025.
Authors' Contact Information: Changxu Liu, State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China; e-mail: liucx22@m.fudan.edu.cn; Hao Zhou, State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China; e-mail: hzhou21@m.fudan.edu.cn; Patrick Dai, Semisand Chip Design Pte. Ltd., Singapore, Singapore; e-mail: patrick@semisand.com; Li Shang, State Key Laboratory of Integrated Chips and Systems, School of Computer Science, Fudan University, Shanghai, China; e-mail: lishang@fudan.edu.cn; Fan Yang, State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China; e-mail: yangfan@fudan.edu.cn.

ACM 1084-4309/2024/08-ART77
https://doi.org/10.1145/3678006

Additional Key Words and Phrases: Multi-scalar multiplication, zero-knowledge proof, privacy-preserving computing, ASIC design

ACM Reference Format:
Changxu Liu, Hao Zhou, Patrick Dai, Li Shang, and Fan Yang. 2024. PriorMSM: An Efficient Acceleration Architecture for Multi-Scalar Multiplication. *ACM Trans. Des. Autom. Electron. Syst.* 29, 5, Article 77 (August 2024), 26 pages. https://doi.org/10.1145/3678006

1 Introduction

Multi-Scalar Multiplication (MSM) is a computationally intensive task with numerous applications in cryptography. It can be regarded as vector inner product operations involving scalars and points on elliptic curves (ECs). It can be expressed as follows:

$$Y = \sum_{i=1}^{N} X_i \cdot G_i. \tag{1}$$

Here, X_i represents a member of scalars, the coefficients of polynomials, and G_i denotes a point on the given elliptic curve. N is the degree of MSM, which refers to the number of scalars and points involved in the computation. It is typically a power of 2 determined by the specific applications. In previous works, MSM can be decomposed into multiple individual scalar multiplications (SMs), each conducted using the double-and-add method [19], followed by summing their results. However, with the emergence of increasingly practical applications in recent years, the value of N is growing larger, at times reaching as high as 2^{26} or even beyond. As a result, how to handle MSM with a larger N is becoming a topic worth discussing. The bucket method [16, 18, 29] has proven to be more efficient than the naive method mentioned above. This advancement has greatly accelerated the implementation speed of MSM, thereby significantly enhancing its overall efficiency.

1.1 MSM in Zero-knowledge Proof

Zero-knowledge proof (ZKP) [15] is a commonly used cryptographic protocol widely applied in blockchain applications and secure multiparty computation. It allows one party (the prover) to prove to another party (the verifier) that the prover knows the secret without revealing any valuable information. In specific applications, ZKP can be utilized for safeguarding patients' private health data [22], ensuring secure authentication for car charging [13], and various other use cases.

ZKP schemes generally involve three stages: *Set up*, *Prove*, and *Verify*. The prover is required to perform extensive computations, involving Number Theory Transformations (NTTs) and MSM, to generate the proof. Upon receiving the proof, the verifier must validate its correctness without requiring additional information from the prover.

Let's consider a simple example: Aleo ZK Proof of Work [28]. In the *Setup* step, it is necessary to prepare a set of points in Lagrange form, $[G]_{SRS}^{eval} = \{G_1, G_2, ...G_{2n}\}$, and a random polynomial in the evaluation form, $c_{2n}^{eval} = \{c_1, c_2, \ldots, c_{2n}\}$. During the *Prove* step, the BLAKE algorithm generates coefficient representations for a corresponding scalar sequence based on a random nonce input. Performing a 2n-size NTT allows us to obtain the evaluation representations of this scalar sequence, denoted as f_{2n}^{eval}. We can further calculate $r_{2n}^{eval} = f_{2n}^{eval} \odot c_{2n}^{eval}$. By executing MSM, we obtain $g_{comm} = (r_{2n}^{eval}) \cdot [G]_{SRS}^{eval}$. Here, r_{2n}^{eval} represents the scalars in MSM, corresponding to X_i in Equation (1). $[G]_{SRS}^{eval}$ denotes the points in MSM, corresponding to G_i in Equation (1). From the *Prove* step, it is evident that the primary computations are concentrated in the NTT and MSM. Notably, the calculations involved in MSM constitute more than 70% of the overall computational overhead [26]. Moreover, the idea of ECNTT [28] suggests that precomputation may reduce the

Table 1. MSM in Various ZKP Protocols [35]

Protocol	Groth16	Plonk	Marlin
Num. of $\mathcal{G}_1$	2	9	6
Num. of $\mathcal{G}_2$	1	0	0
MSM in proof time	70%–80%	85%–90%	70%–80%

[1]The exact value depends on a specific application.

need for frequent use of NTT. Nevertheless, it is important to emphasize that MSM retains its essential role, highlighting the ongoing significance of MSM in ZKP schemes.

There are several other representative ZKP schemes, including zk-SNARK [6], Groth16 [17], BulletProofs [9], Plonk [14], and Marlin [10]. All of the above protocols rely on MSM to compute polynomial commitments [35]. In Table 1, it is evident that certain representative protocols allocate a significant portion of their processing time to MSM computations. Consequently, accelerating MSM computations is of crucial significance for ZKP applications and remains a primary focus of our attention. We firmly believe that the advancements in MSM hold immense potential for enhancing numerous real-world applications.

1.2 Related Works

Previous works have made significant optimizations and improvements for operations on elliptic curves. Enhancements in basic multiplier circuits have been achieved. The work [33] presents efficient VLSI designs for optimized radix multiplication in GF(2^m). These designs include bit-serial, digit-serial, and bit-parallel structures. Additionally, there are custom hardware designs for point multiplication tailored to specific curves. The work [32] introduces efficient hardware implementations for binary Edwards curves. These implementations utilize novel complete differential addition formulas to enhance the speed and efficiency of point multiplication operations. The work [24] introduces a novel high-speed ECC processor, leveraging segmented pipelined full-precision multipliers and a modified LD Montgomery point multiplication algorithm, achieving the fastest ECC processor design on FPGA at that time. The survey [31] provides a comprehensive summary of advancements in hardware implementations of ECC, covering various elliptic curves, point multiplication algorithms, and finite field arithmetic discussions, along with a classification comparison and performance evaluation of FPGA and ASIC implementations.

While MSM is based on ECC, its computational essence is fundamentally different, predominantly consisting of a large number of point multiplication operations. Many outdated hardware solutions rely on traditional point multiplication units [5, 30, 34] that are implemented using the double-and-add scheme. Extending these schemes to a larger degree is straightforward but naive. Data dependencies result in underutilized computational units, making the processing of MSM with a larger degree intolerably time-consuming. To further speed up the implementation of MSM, some works underwent algorithmic innovations. The works [16] and [18] have introduced some ideas on how to speed up the implementation of MSM. While the bucket method is easy to understand, we aim to investigate some optimizations and its efficient implementation on hardware applications.

Some works aim to facilitate the implementation of ZKP including MSM by constructing universal libraries. *gnark* [8] is a zk-SNARK library that provides a high-level API for designing circuits. It supports Groth16 and Plonk. *EdMSM* [21], built upon *gnark-crypto*, is implemented in Go and uses hand-written arm64 assembly to accelerate the MSM. Nonetheless, given the inherently parallelizable nature of MSM, it does not align optimally with the computational strengths of CPUs, leading to suboptimal implementation efficiency.

Naturally, researchers try to explore leveraging GPUs to accelerate the implementation of MSM, given their inherent parallelism. *GZKP* [27] and *cuZK* [26] represent GPU-accelerated ZKP, with a primary focus on MSM. While utilizing bucket methods, they prioritize aligning MSM implementations with GPU architecture, lacking a thorough exploration of MSM's intrinsic mathematical properties. Despite the low latency of GPU-based MSM computation, it incurs high power consumption without achieving an optimal tradeoff. Moreover, finite field-based MSM underutilizes GPU floating-point processing units, leading to performance redundancy. *Elastic MSM* [38] is a recent development in MSM that focuses on leveraging precomputation techniques to utilize the computational power of GPUs fully. It significantly enhances performance while imposing considerable demands on device storage capacity and bandwidth.

Given limitations on general-purpose platforms, some works have focused on achieving optimal MSM performance through custom circuits, primarily utilizing ASIC and FPGA implementations. *PipeZk* [36] introduces an end-to-end pipelined design for zk-SNARK on ASIC. However, the design principles of the Buckets in the MSM limit potential improvements in processing efficiency. Additionally, a substantial amount of data is transferred back to the host for computation, and *PipeZk* cannot handle point double (PDBL). Consequently, we argue that it does not qualify as a comprehensive MSM accelerator. *PipeMSM* [35] and *CycloneMSM* [2] are FPGA implementations of MSM. However, due to the data dependencies in the pipeline caused by the mixadd-based point addition (PADD) design, its data scheduling mechanism continues to introduce numerous pipeline bubbles, thereby compromising performance. Additionally, the bucket aggregation algorithm employed by *CycloneMSM* is relatively simple, potentially creating another bottleneck in MSM with slightly smaller degrees. *Gypsophila* [25] proposes a scalable and bandwidth-optimized MSM architecture, primarily focusing on the optimization of the multi-MSM hardware architecture, without making significant optimizations to the bucket method, particularly the bucket aggregation step. *BSTMSM* [37], based on the Xilinx U250 platform, is a recently released MSM accelerator. It primarily focuses on reducing bucket access collisions to improve pipeline efficiency. This aligns with the goals of our work; however, its design employs a significant amount of true dual-port SRAM as the Buckets, resulting in a much larger storage overhead compared to our approach. From an application perspective, it is not practical.

1.3 Our Contributions

We have identified several issues in the previous works:

There is further potential for improving the computational efficiency of the bucket classification step. While the performance of the MSM accelerator mainly relies on the PADD units, the utilization of PADD in previous works was not optimal. For instance, the mixadd-based PADD unit introduces data dependencies in the MSM computational pipeline, thereby limiting performance. In an ideal scenario, if the MSM accelerator has only one pipeline PADD unit and we fully exploit its computational capacity, the average number of clock cycles per point should be around 1. However, previous works result in higher clock cycle counts per point. This observation confirms our findings.

They have almost completely overlooked optimizing the bucket aggregation step in their implementation. When the degree of the MSM is relatively small, the approaches taken in previous works greatly limit the overall performance of the MSM due to the implementation of this step. Additionally, at this point, the utilization of PADD units is low, and their computational performance is not fully realized.

There should be a proper guideline for selecting the window size. The selection of window size is critical in a custom circuit-based solution. However, in previous works, the selection of the window size is arbitrary or lacks rationale. In some cases, they opt for a larger window size to reduce the time consumption of the bucket classification step, but this results in a significant increase in the area overhead of the bucket and the time consumption of the bucket aggregation step.

To address these performance limitations of previous MSM accelerators or achieve a more balanced tradeoff between area and speed, in this article, we propose PriorMSM, a highly efficient acceleration architecture for MSM. Our architecture can accommodate any elliptic curve choice; however, there may be variations in the design of the PADD unit. The selection of curves should be based on a comprehensive evaluation of security and performance tailored to the specific applications. We divide the MSM into three steps: bucket classification, bucket aggregation, and result aggregation. Each of these steps has undergone improvements and optimizations to enhance their performance. Our contributions include:

— We employ multi-FIFO and multi-bank architecture to construct a Priority-Based Scheduling Mechanism (PBSM) that improves the pairing success rate of points, minimizing stalls and bubbles in the pipeline of MSM. This architecture achieves an average clock cycle per point close to 1 in the bucket classification step of MSM. We utilize precomputation techniques and the ωNAF method to compress the effective scale of the index used in the bucket method, reducing the on-chip SRAM area to just 37.5% of its original size.
— We further propose a highly efficient implementation and in-depth analysis of the bucket aggregation step, which prior works have ignored. A partitioning algorithm is introduced to enable the "parallel" execution of the bucket aggregation step. By leveraging the inherently fully pipelined nature of PADD, this partitioning algorithm can significantly enhance the performance of the bucket aggregation step on hardware. Experimental results demonstrate that this optimization can reduce the time consumption of this step by nearly 95%.
— We analyze the design sensitivity of MSM, investigating the impact of optimizing the bucket aggregation step on the overall performance of MSM. Additionally, we examine the influence of window size selection on its performance. We do not blindly pursue an increase in window size, even though it significantly improves the implementation speed of MSM. Instead, we consider using the Area-Time Product (ATP) metric to evaluate the relative optimality of MSM design parameters, providing valuable insights for subsequent works.

The remainder of the article is structured as follows. In Section 2, we provide foundational concepts related to elliptic curves. Section 3 presents the details of PriorMSM. We provide a mathematical description of the bucket method and the optimized bucket aggregation. This section provides a comprehensive introduction to the submodule design and overall architecture of PriorMSM. We also elucidate employed strategies, such as precomputation techniques and the ωNAF method for scalar processing, contributing to hardware optimization. Section 4 unveils the implementation of our design, detailing adopted parameters. Tables and figures effectively demonstrate our comprehensive exploration of design sensitivity and the performance evaluation of our design. Additionally, this section includes a comparison of PriorMSM with previous works. In Section 5, we conclude with remarks on our work. Appendix A presents a more detailed theoretical analysis of the latency of the bucket method employed in this article, as well as the effectiveness of our optimization techniques for the bucket aggregation algorithm.

2 Preliminary

An EC is a cubic curve whose solutions are confined to a region of space topologically equivalent to a torus. To facilitate understanding, we'll explain the concept of elliptic curves from an engineering perspective rather than providing a rigorous mathematical definition. Simply put, an elliptic curve E over F_q is the set of points that satisfy Equation (2), where coefficients a_{sw} and b_{sw} must satisfy a certain condition $4a_{sw}^3 + 27b_{sw}^2 \neq 0$. This is commonly known as the general Short Weierstrass form of the elliptic curve.

$$y^2 = x^3 + a_{sw}x + b_{sw}. \tag{2}$$

Additionally, various types of elliptic curves exist, such as the Edwards curve, Hessian curve, Montgomery curve, Twisted Edwards curve, and Twisted Hessian curve. Each curve is defined by a distinct formula; for instance, the Montgomery curve can be expressed by Equation (3). The Twisted Edwards curve can be represented as Equation (4).

$$B_{mon}y^2 = x^3 + A_{mon}x + x. \tag{3}$$

$$a_{te}x^2 + y^2 = 1 + d_{te}x^2y^2. \tag{4}$$

Many elliptic curves are birationally equivalent to each other or satisfy certain special conditions that establish equivalence. For example, any elliptic curve can be transformed into the Short Weierstrass form through a birational equivalence [1]. Each Twisted Edwards curve can be transformed into an equivalent Montgomery curve through birational equivalence. In particular, the Twisted Edwards curve $E_{a_{te},d_{te}}$ is birationally equivalent to the Montgomery curve $M_{A_{mon},B_{mon}}$. The transformation of parameters and the mapping from a point (x, y) on curve $E_{a_{te},d_{te}}$ to a point (x', y') on curve $M_{A_{mon},B_{mon}}$ is expressed by the following equation:

$$\begin{cases} A_{mon} &= \frac{2(a_{te}+d_{te})}{a_{te}-d_{te}}, \\ B_{mon} &= \frac{4}{a_{te}-d_{te}}, \\ (x', y') &= \left(\frac{1+y}{1-y}, \frac{1+y}{x(1-y)}\right). \end{cases} \tag{5}$$

Similarly, when mapping from the Montgomery curve $M_{A_{mon},B_{mon}}$ to the Short Weierstrass curve $E_{a_{sw},b_{sw}}$, the transformation relationship is depicted by Equation (6), where (x'', y'') represents a point on the curve $E_{a_{sw},b_{sw}}$:

$$\begin{cases} a_{sw} &= \frac{3-A_{mon}^2}{3B_{mon}^2}, \\ b_{sw} &= \frac{2A_{mon}^3-9A_{mon}}{27B_{mon}^3}, \\ (x'', y'') &= \left(\frac{x'}{B_{mon}} + \frac{A_{mon}}{3B_{mon}}, \frac{y'}{B_{mon}}\right). \end{cases} \tag{6}$$

The transformation from the Short Weierstrass curve to the Twisted Edwards curve is the inverse of the transformation mentioned above. For further details, refer to [11].

In addition to the various curve forms, each curve can possess different coordinate representations. In the above equations, x and y are in affine coordinates, indicating that (x, y) represents a point on the curve. Additionally, there can be projective coordinate representations, where (X, Y, Z) represents a point on the curve. A point (x, y) in affine coordinates is equivalent to $(X/Z, Y/Z)$. These are just the more common forms of coordinate representations; other forms exist, such as extended coordinates under the Twisted Edwards curve [20]. By introducing an auxiliary coordinate $t = xy$, we can use (x, y, t) in extended affine coordinates to represent (x, y) in affine coordinates.

In Equation (1), G_i represents an element in an EC denoted as E, which consists of a total of N points. The operation referred to as "multiplication" between X_i and G_i is commonly recognized as point multiplication (PMUL) or scalar multiplication (SM). Typically, this operation is transformed into a series of PADD operations and can be articulated as follows:

$$X_i \cdot G_i = \underbrace{G_i + G_i + \cdots + G_i}_{X_i\ times}. \tag{7}$$

The outcome of scalar multiplication is a point on E. When multiple scalar multiplications are carried out and their results are collectively accumulated, the outcome remains a point on E.

Table 2. Comparison of Different Coordinate Systems

Coordinates	Affine coordinates	Projective coordinates	Extended coordinates	Inverted coordinates
Num. of mul	7	13	9	12
Num. of inv	2	0	0	0
Num. of add	4	7	9	7
Unified	Yes	Yes	Yes	Yes

To provide further clarification, let's consider the example of BLS12-377 [4]. BLS12-377 is a widely employed elliptic curve in cryptography, particularly for applications such as ZKPs and other cryptographic schemes. In our context, the BLS12-377 curve serves as the primary target for PriorMSM design considerations. However, it's essential to highlight that PriorMSM is an architecture not inherently tied to a specific curve. BLS12-377 over G_1 can be mathematically expressed in the Short Weierstrass form as follows:

$$y^2 = x^3 + 1. \tag{8}$$

As previously mentioned, many curves, including widely used ones such as BLS12-377, BLS12-381, and BN254, exhibit birational equivalence [7, 11]. Consequently, BLS12-377 can also be represented in the Twisted Edwards curve as illustrated in Equation (4), where

$$\begin{aligned} a_{te} &= -1. \\ d_{te} &= 136396142414293534522166394536258004439411625840037520960 \\ &\quad 3501090846867915629550320449265247983373243775153605550 12. \end{aligned} \tag{9}$$

An appealing characteristic of the Twisted Edwards curve lies in its strongly unified point addition law. A unified point addition formula eliminates the need to handle exceptions in specific scenarios, such as adding two identical points or adding a point at infinity. Simultaneously, the Twisted Edwards curve's addition formula boasts faster computation than unified formulas in alternative curve forms. Its notable efficiency stems from the strong property of completeness, enhancing its performance in cryptographic protocols. For additional details, refer to [20]. So in our proposed design, we construct a PADD unit based on the Twisted Edwards curve.

When it comes to coordinate system selection, both PADD and PDBL can be constructed using several modular multiplications (**mul**), modular additions (**add**), and modular inverses (**inv**). Referring to [20] and [12], we summarize the counts of these operations in BLS12-377 across four primary coordinate systems. Details are presented in Table 2. The main factors influencing performance are **mul** and **inv**. Inevitably, implementing **inv** in affine coordinates can be inefficient. In PriorMSM, we leverage the extended projective coordinates under the Twisted Edwards curve, enabling the implementation of a unified PADD with only 9 **mul**s.

3 Proposed Design

In this section, we delve into the hardware acceleration scheme designed for MSM. The foundation of our approach lies in the meticulous design of the PADD, an indispensable component in the MSM algorithm, as expounded in Section 3.1. The PADD unit is tasked with performing the addition of two points on an elliptic curve, and we present a comprehensive overview of its hardware implementation. Moving forward, Section 3.2 outlines the algorithm forming the basis of our work, along with the optimizations we have implemented. This includes a detailed exploration of bucket classification and aggregation steps. The subsequent Section 3.3 delves into the hardware implementation specifics of PriorMSM, featuring a highly efficient PBSM. Section 3.4 introduces the ωNAF method and precomputation techniques. These prove instrumental in significantly

Table 3. Four-processor Twisted Edwards Unified Addition

Cost	Step	Thread 1	Thread 2	Thread 3	Thread 4
2 **muls**	**1**	idle	idle	$R_1 \leftarrow Z_1 * Z_2$	$R_2 \leftarrow T_2 * k$
5 **adds**	**2**	$R_3 \leftarrow Y_1 + X_1$	$R_4 \leftarrow Y_1 - X_1$	$R_5 \leftarrow Y_2 + X_2$	$R_6 \leftarrow Y_2 - X_2$
3 **muls**+1 **add**	**3**	$R_7 \leftarrow R_3 \cdot R_5$	$R_8 \leftarrow R_4 \cdot R_6$	$R_9 \leftarrow 2 \cdot R_1$	$R_{10} \leftarrow T_1 \cdot R_2$
4 **adds**	**4**	$R_{11} \leftarrow R_7 + R_8$	$R_{12} \leftarrow R_7 - R_8$	$R_{13} \leftarrow R_9 + R_{10}$	$R_{14} \leftarrow R_9 - R_{10}$
4 **muls**	**5**	$X \leftarrow R_{12} \cdot R_{14}$	$Y \leftarrow R_{11} \cdot R_{13}$	$Z \leftarrow R_{13} \cdot R_{14}$	$T \leftarrow R_{11} \cdot R_{12}$

curtailing the on-chip storage circuit's spatial requirements. We also explore the seamless integration of these techniques into our hardware implementation of the MSM algorithm.

3.1 PADD Unit

The computational power of MSM is primarily provided by the PADD unit. Remarkably, the PADD unit is tasked with two distinct operations: the first is point addition, which entails the combination of two not-equal points, and the second is point double, a process that involves the addition of two identical points. In addressing both scenarios, a unified formula [20] can be adeptly applied.

In Section 2, we introduce the utilization of extended Twisted Edwards coordinates for representing points. The addition formula can be referenced in Section 3.1 in [20]. Table 3 outlines the Four-processor Twisted Edwards unified point addition, specifically tailored for BLS12-377.

Following the steps outlined in Table 3, we design a fully pipelined PADD unit that necessitates a total of 9 **mul**s and 9 **add**s. The PADD unit described in [35] necessitates 7 **mul**s, with the caveat that it takes a duration of two cycles to generate a point. The pipeline feature of PADD is crucial for accelerating the implementation of MSM, with both two-cycle and one-cycle pipelines exerting significant influence on MSM calculations. To optimize performance with the PADD unit, we employ full unrolling of the point addition formula. This results in a highly efficient PADD unit capable of concurrently processing two input points and generating a single-output result per cycle. The pipelined data path of this unit is depicted in Figure 1. The inputs (X_1, Y_1, Z_1, T_1) and (X_2, Y_2, Z_2, T_2) are in extended projective coordinates, and the output of PADD is denoted by $(X_{res}, Y_{res}, Z_{res}, T_{res})$.

The high-cost pipelined **mul** incurs a longer latency compared to the **add** operation, thus dominating the overall computation time. We employ the Barrett reduction algorithm from [35] and implement a four-layer Karatsuba method [23] to construct the high-cost pipelined **mul**. Each **mul** involves a set of 24-bit multipliers. Our PADD unit achieves an operating frequency of up to 1 GHz in the TSMC 28 nm process. A single **mul** requires 27 cycles, resulting in a total latency of 87 cycles for the PADD unit.

3.2 Optimized Bucket Method

The bucket aggregation step in the bucket method is often overlooked, leading to the adoption of a naive approach that is often inefficient and has low hardware utilization. By leveraging the advantages of custom hardware design for pipelining, we develop a highly efficient bucket aggregation step, resulting in an optimized bucket method.

To facilitate comprehension, we provide the symbols outlined in Table 4. The bucket method, also Pippenger's algorithm, involves dividing a b-bit scalar into smaller c-bit segments, referred to as indexes. In detail, each scalar can be represented as $(a_{i(m-1)}, \ldots, a_{ij}, \ldots, a_{i1}, a_{i0})$, where each a_{ij} is a c-bit integer. We refer to MSM between each group of indexes and points as the reduced MSM. Therefore, our focus can be solely on the reduced MSMs since they are independent. By aggregating their computation results, we can derive the overall outcome of the MSM. Furthermore, MSM can be represented by the following three steps, as illustrated in Figure 2:

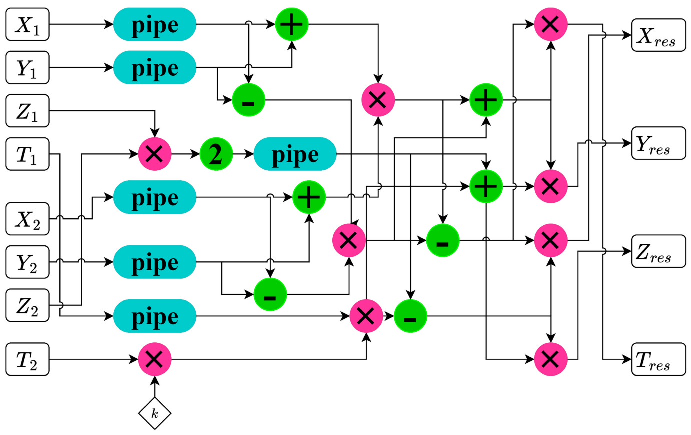


Fig. 1. The fully pipelined PADD unit in extended coordinates. The **mul** is denoted by a red circle with an "x". The **add** is represented by a green circle with a "2" or "+". The modular subtraction unit can also be represented by an **add** and is indicated by "-". The buffer used to align data is shown as a cyan "pipe." k is a parameter, and it is defined as $k = 2d_{te}$.

Table 4. Symbols and Descriptions in Section 3

Symbol	Description
N	Degree of MSM
b	Bit width of a scalar
c	Bit width of reduced scalar or window size
a	Reduced scalar or index (each one has width with c-bit)
G	Point on elliptic curve
S	Buckets. When there are two subscripts, the former represents the index of the reduced MSM and the latter represents the index of the bucket. When there is only one subscript, it only represents the index of the bucket.
m	$\lceil \frac{b}{c} \rceil$, the number of reduced MSMs
k	Value of index, $k = a_{ij}$
H	Number of groups of buckets (lowercase for the index of the current group)
M	$\lceil \frac{2^c}{H} \rceil$, number of buckets each group
D	Number of clock cycles required for one PADD unit

— Bucket classification
— Bucket aggregation
— Result aggregation

Mathematically, these three steps can be expressed as Equation (10):

$$\begin{cases} S_{jk} & = \sum_{i=1}^{N} (a_{ij} == k) \cdot G_i, \\ R_j & = \sum_{i=1}^{N} a_{ij} \cdot G_i = \sum_{k=0}^{2^c-1} k \cdot S_{jk}, \\ Y & = \sum_{i=1}^{N} X_i \cdot G_i = \sum_{i=1}^{N} \sum_{j=0}^{m-1} 2^{jc} \cdot a_{ij} \cdot G_i, \\ & = \sum_{j=0}^{m-1} \sum_{i=1}^{N} 2^{jc} \cdot a_{ij} \cdot G_i = \sum_{j=0}^{m-1} 2^{jc} \cdot R_j. \end{cases} \tag{10}$$

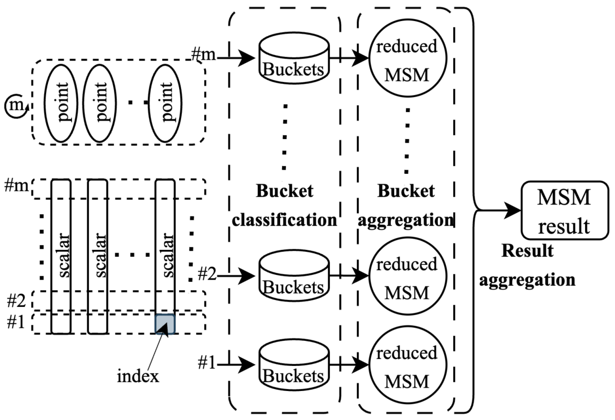


Fig. 2. We divide the MSM into m reduced MSMs, consisting of three steps. The first step is bucket classification, responsible for classifying points into corresponding buckets based on their indexes. The second step is bucket aggregation, where the values contained in each reduced MSM's buckets are aggregated. The final step is result aggregation, where the results from each reduced MSM are aggregated together.

S_{jk} represents the bucket classification. We begin by classifying distinct G_i into bucket k based on their corresponding indexes, a_{ij}, where $k = a_{ij}$ and $0 \leq k < 2^c$. R_j represents the bucket aggregation, where we calculate the sum of all points within buckets. Each point is multiplied by its corresponding index value k. After aggregating the outcomes of each reduced MSM, R_j, by multiplying them with a factor of 2^{jc}, we obtain Y, which represents the final result of the MSM. This aggregation aligns the computation result of the jth reduced MSM with the position of its c-bit scalar within the b-bit scalar. Algorithm 1 outlines the details of the bucket method.

We observe that numerous studies employing the bucket method have often overlooked opportunities for enhancing the bucket aggregation step and leveraging the pipelining capabilities of the PADD unit. Due to the fixed computation paradigm of the bucket aggregation step, the data dependency relationships can potentially limit the performance of calculations. In more severe cases, when N is smaller and c is larger, or in other words, when $\frac{\log_2(N)}{c}$ decreases, accelerating the bucket aggregation step becomes increasingly crucial for MSM.

In Algorithm 1, the naive bucket aggregation in reduced MSM can be expressed as

$$\begin{aligned}\sum_{k=1}^{2^c-1} kS_k &= (2^c-1)\cdot S_{2^c-1} + (2^c-2)\cdot S_{2^c-2} \\ &\quad + \cdots + 2\cdot S_2 + S_1.\end{aligned} \tag{11}$$

In this step, the algorithm is only dependent on the values of c and those in S, and it is no longer influenced by N. In this computation paradigm, we need to sequentially traverse $2^c - 1$ bucket values, which constitutes a very long computational chain.

By introducing a partitioning algorithm, we exploit the parallelism of the bucket aggregation step. We partition all the buckets into H groups, with each group comprising M points. Typically, H is chosen as a power of 2. Equation (11) can be expressed as follows:

$$\begin{aligned}\sum_{k=1}^{2^c-1} kS_k &= [(H*M-1)\cdot S_{H*M-1} + (H*M-2)\cdot S_{H*M-2} \\ &\quad + \cdots + ((H-1)*M+1)\cdot S_{(H-1)*M+1}]\end{aligned}$$

ALGORITHM 1: Bucket method

```
 1: Function: Bucket Classification
 2: set $S$ = NULL (Point at infinity)
 3: for $j = 0, j < m, j++$ do
 4:     for $i = 0, i < N, i++$ do
 5:         set $k = a_{ij}$                                ▷ Identify the target bucket.
 6:         $S_{j,k} \leftarrow S_{j,k} + G_i$
 7:     end for
 8: end for
 9: Function: Bucket Aggregation
10: for $j = 0, j < m, j++$ do
11:     set $R_j$ = NULL (Point at infinity)
12:     for $k = 2^c, k > 0, k--$ do
13:         $R_j \leftarrow R_j + S_{j,k}$
14:         $S_{j,k-1} \leftarrow S_{j,k} + S_{j,k-1}$
15:     end for
16: end for
17: Function: Result Aggregation
18: set $Y$ = NULL (Point at infinity)
19: for $j = m-1, j > 0, j--$ do
20:     $Y \leftarrow 2^c \cdot Y + R_j$
21: end for
```

$$\begin{aligned}
&+ \dots \\
&+ [(h * M) \cdot S_{h*M} + (h * M - 1) \cdot S_{h*M-1} \\
&+ \dots + ((h-1) * M + 1) \cdot S_{(h-1)*M+1}] \\
&+ \dots \\
&+ [(M) \cdot S_M + (M-1) \cdot S_{M-1} + \dots + S_1].
\end{aligned} \tag{12}$$

For each group out of them, it can be expressed as

$$\begin{aligned}
&(h * M) \cdot S_{h*M} + (h * M - 1) \cdot S_{h*M-1} \\
&+ \dots + ((h-1) * M + 1) \cdot S_{(h-1)*M+1}.
\end{aligned} \tag{13}$$

We can rewrite it as

$$\begin{aligned}
&[(M) \cdot S_{h*M} + (M-1) \cdot S_{h*M-1} + \dots + S_{(h-1)*M+1}] \\
&+ (h-1) * M * [S_{h*M} + S_{h*M-1} + \dots + S_{(h-1)*M+1}].
\end{aligned} \tag{14}$$

Thanks to the fully pipelined PADD unit, the computation of the first term in Equation (14) can be executed in the form of "parallel." Here, "parallelism" does not strictly imply complete parallelism but rather signifies that the bucket in each group can be queued for entry into the PADD unit. The calculations are carried out independently but in a temporally compact manner.

We employ a straightforward example with $H = 4$ and $M = 4$ to illustrate the computation of partitioned bucket aggregation using a fully pipelined PADD unit. The process is illustrated in Figure 3. To derive the result of the bucket aggregation step (denoted as "Target" in Figure 3), in *Step* 1, PADD computes R and T by using the method described in Algorithm 1. However, since the buckets are divided into four groups, this process can be executed "in parallel" using pipelined PADD. In *Step* 2 and *Step* 3, we aggregate these groups to obtain the target. We can utilize the

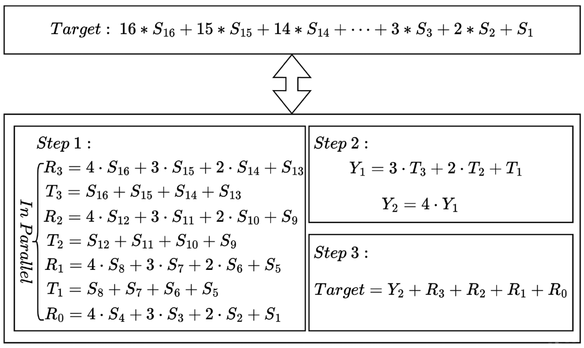

$Target: \; 16 * S_{16} + 15 * S_{15} + 14 * S_{14} + \cdots + 3 * S_3 + 2 * S_2 + S_1$
$Step\ 1:$
In Parallel
$R_3 = 4 \cdot S_{16} + 3 \cdot S_{15} + 2 \cdot S_{14} + S_{13}$
$T_3 = S_{16} + S_{15} + S_{14} + S_{13}$
$R_2 = 4 \cdot S_{12} + 3 \cdot S_{11} + 2 \cdot S_{10} + S_9$
$T_2 = S_{12} + S_{11} + S_{10} + S_9$
$R_1 = 4 \cdot S_8 + 3 \cdot S_7 + 2 \cdot S_6 + S_5$
$T_1 = S_8 + S_7 + S_6 + S_5$
$R_0 = 4 \cdot S_4 + 3 \cdot S_3 + 2 \cdot S_2 + S_1$
$Step\ 2:$
$Y_1 = 3 \cdot T_3 + 2 \cdot T_2 + T_1$
$Y_2 = 4 \cdot Y_1$
$Step\ 3:$
$Target = Y_2 + R_3 + R_2 + R_1 + R_0$

Fig. 3. A simple partitioned bucket aggregation with $H = 4$ and $M = 4$.

acquired T values to replace the original longer chain of PADD operations with a shorter chain, reducing the number of PADD operations needed.

Optimization rate is introduced as a metric to evaluate the effectiveness of our partitioning algorithm on the bucket aggregation algorithm. It can be expressed by the following formula:

$$\textit{optimization rate} = \frac{\text{clock cycles consumed by the optimized bucket aggregation algorithm}}{\text{clock cycles consumed by the naive bucket aggregation algorithm}}. \tag{15}$$

The mathematical representation is given by Equation (22) in Appendix A. A lower *optimization rate* indicates a better optimization effect. This algorithm optimization represents a significant advancement, greatly reducing the number of clock cycles required during the bucket aggregation step. For instance, with $c = 16$ and $H = 16$, the *optimization rate* is about 6.27%. A detailed mathematical analysis of the optimized bucket aggregation algorithm and the calculation method for the *optimization rate* are presented in Appendix A.

3.3 Architecture

The proposed acceleration architecture, PriorMSM, which implements the bucket method described in Section 3.2, is illustrated in Figure 4. PriorMSM consists of several key components: the scheduler, the computation module (PADD), the storage modules (FIFO, Buckets, and Flag registers indicating the validity of data in Buckets), and various data switching and selection modules (Crossbar and Mux). External data is fetched from the DRAM and then directed into the PriorMSM. On the right side of Figure 4, we present the microarchitecture of the scheduler, which primarily consists of Flag registers and the Bucket R&W controller, FIFO R&W controller, PADD controller, and Process controller. The arrows in the diagram indicate the direction of information flow. Flag registers record the validity of the data in the Bucket and provide information to the Bucket's R&W controller. The Bucket R&W controller manages the read/write actions on the Bucket, generates addresses, and notifies the FIFO R&W controller if a read operation occurs. The FIFO R&W controller oversees all read/write actions on the FIFOs, including selecting the FIFO being operated on and monitoring the status of the FIFO. The PADD controller, relatively simple in design, primarily controls the start/stop of the PADD and monitors for valid data in the PADD pipeline. The Process controller monitors the status information of each controller to control the overall calculation process of PriorMSM. In other words, it integrates the scheduling mechanism, PBSM, proposed by us.

Bucket Classification: The acceleration of MSM primarily hinges on the bucket classification step, a focal point of our hardware design. Leveraging multi-FIFO and multi-bank architecture, we propose the PBSM. PBSM augments the likelihood of pairing points, thereby minimizing pipeline

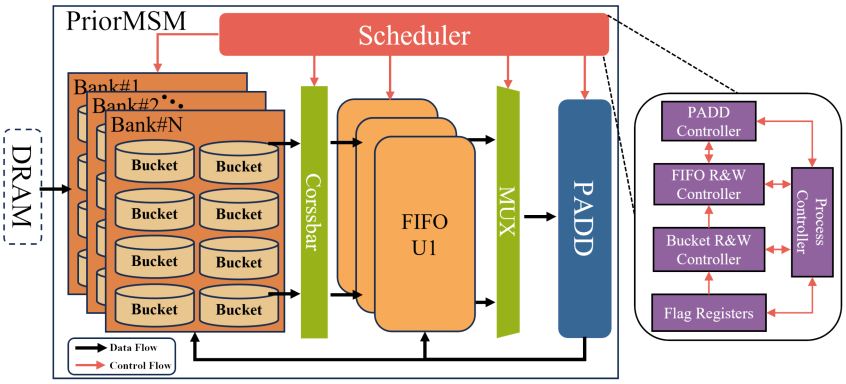


Fig. 4. Architecture of PriorMSM. On the right side is the microarchitecture of the Scheduler, with arrows indicating the direction of information flow.

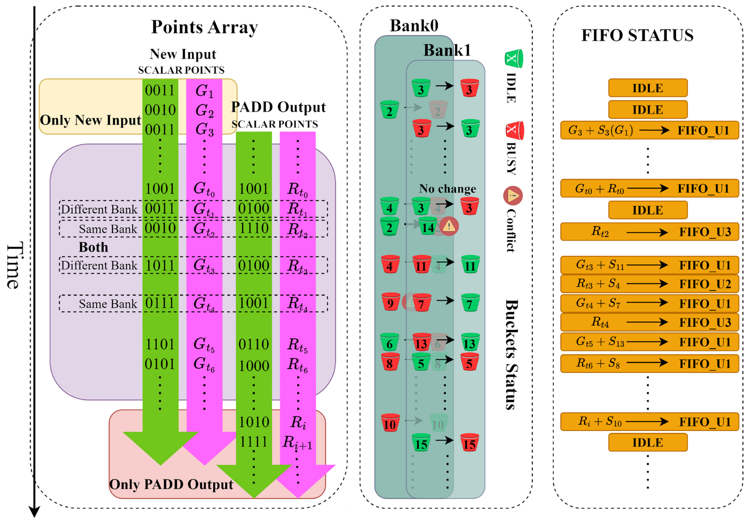


Fig. 5. Data scheduling mechanism of PriorMSM. We choose $c = 4$ for simplicity and implement the Buckets using two banks.

stalls and ensuring a high-throughput data flow in the PADD. We illustrate PBSM with a simple demo in Figure 5. Window size is set to 4 and Buckets are implemented using SRAM with two banks.

In the initial stage, the pipeline of PADD is empty. Consequently, only new input data from the external DRAM is utilized, with the corresponding a_{ij} serving as an index. If there is no point in the Buckets with the associated index, the bucket flag is set to valid, and the new input point is directly written into the corresponding bucket, as shown by G_2 in Figure 5. If a point already exists in the Buckets, both the previously stored point S_3 (corresponding to the earlier G_1) and the new input point G_3 are transferred to FIFO_U1. Subsequently, the bucket flag is reset to invalid.

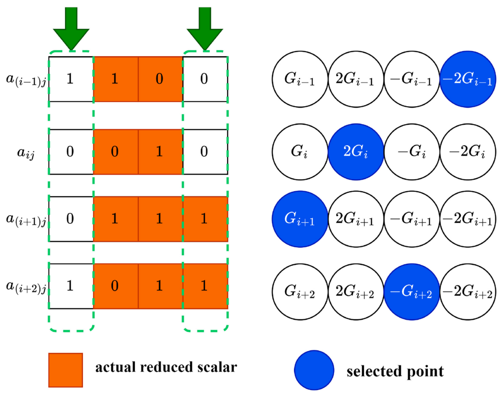
PADD reads pairs of points from FIFO_U1, and upon completing the computationally intensive point addition, it produces the results. Most of the time, PADD continuously generates results that can be either written back into the Buckets or sent to the FIFOs, along with existing points from the Buckets. This process corresponds to the stage denoted as "Both" (the block highlighted in purple) in Figure 5. An ideal situation occurs when the index of the new input point is equal to the index of the PADD output point, enabling them to be sent into FIFOs even if the corresponding position in Buckets has valid data, as exemplified by G_{t0} and R_{t0}. This strategy helps to minimize the dynamic power consumption associated with reading and writing to Buckets. Alternatively, in certain cases, one of both the input point and the PADD output point may need to be written into the Buckets. At the same time, the other is sent to the FIFO alongside a corresponding existing point in the Buckets. This is illustrated in scenarios such as G_{t5} and R_{t5} or G_{t6} and R_{t6}. We employ SRAM instead of the registers used in [36] for the Buckets, which is advantageous for optimizing circuit area while also supporting wider window sizes. However, due to the structure of the Buckets, it cannot concurrently read or write multiple points, resulting in conflicts with the points provided by the external DRAM. This conflict can occur when both points need to be written into the Buckets (G_{t1} and R_{t1}, G_{t2} and R_{t2}) or paired with points in the Buckets (G_{t3} and R_{t3}, G_{t4} and R_{t4}). Interrupting the data flow from DRAM to accommodate PADD data can result in a significant number of pipeline bubbles in the PADD pipeline, significantly impacting computation speed.

Minimizing pipeline bubbles in PADD is crucial for accelerating MSM. This necessitates enhancing the success rate of point pairing. To achieve this objective, we design the Buckets using a multi-bank structure that allocates points to different banks based on the low bits of the indexes. Thus, if PADD output points and external input points belong to different banks, they can be independently accessed from their respective banks. In Figure 5, both G_{t1} and R_{t1} need to be written into the Buckets, as do G_{t3} and R_{t3}, which also require pairing with points in the Buckets. Utilizing multi-FIFOs assists in efficiently storing the paired points, rather than immediately pushing them into the PADD for processing. FIFO_U2 is introduced to receive the new pair of points allowed under the multi-bank structure. If the point output from PADD and the external input point belong to the same bank and they need to be read or written, respectively, this is supported and the corresponding point will be written to Buckets or paired with the point read from Buckets, respectively. The scenario is identical to that of G_{t5} and R_{t5}, G_{t6} and R_{t6}. However, in scenarios where both points require pairing with Buckets or need to be written to Buckets, conflicts may arise with identical requests, as seen in cases like G_{t2} and R_{t2}, G_{t4} and R_{t4}. In this case, the external input point is preferentially written to the Buckets or paired with the point in the Buckets. The point output from PADD is directed to FIFO_U3 to increase its priority so that it can re-enter the pairing circuit in subsequent cycles. Considering the scenario under “Only PADD Output,” the data scheduling mechanism is the same as that for the “Only New Input” stage, as depicted by R_i and R_{i+1} in Figure 5.

Therefore, PADD retrieves pairs of points from FIFO_U1 and FIFO_U2, aiming to fill its pipeline as much as possible. Points entering FIFO_U3 will be reinserted into the matching queue. The aforementioned explanation pertains to the scenario of a single PADD unit. However, when there are multiple PADD units within an MSM, the PBSM can also be utilized to achieve high throughput in MSM implementations.

Bucket Aggregation: The enhanced algorithm for this step has been thoroughly explained in Section 3.2. We implement the proposed approach in hardware and observe significant improvements compared to the unoptimized version. All S_k values are stored in the Buckets. The corresponding values are transferred from the Buckets to FIFO_U1 for processing in PADD. Subsequently, the computation results from PADD are combined with the corresponding values from the Buckets and resent to FIFO_U1. The fully pipelined PADD unit can efficiently accomplish the bucket aggregation step.

Result Aggregation: Each reduced MSM generates one point, resulting in a total of m results. To consolidate these outcomes into the final result, $m - 1$ PADD and $b - c$ PDBL operations are



$a_{(i-1)j}$
1 1 0 0
a_{ij}
0 0 1 0
$a_{(i+1)j}$
0 1 1 1
$a_{(i+2)j}$
1 0 1 1
G_{i-1} $2G_{i-1}$ $-G_{i-1}$ $-2G_{i-1}$
G_i $2G_i$ $-G_i$ $-2G_i$
G_{i+1} $2G_{i+1}$ $-G_{i+1}$ $-2G_{i+1}$
G_{i+2} $2G_{i+2}$ $-G_{i+2}$ $-2G_{i+2}$
actual reduced scalar
selected point

Fig. 6. By observing the "head" and "tail" of the reduced scalars, the scale of indexes can be compressed to decrease the number of buckets.

required, taking approximately $(b - c + m - 1) \cdot D$ clock cycles. The time required for the result aggregation step in the overall MSM algorithm is practically negligible, and this step is divided to be performed after each bucket aggregation step.

PriorMSM is built on multi-FIFO and multi-bank architecture with a PBSM. This design maximizes the pairing rate of points, thereby minimizing bubbles in the PADD pipeline. According to the analysis in Section 3.2, the number of PADD operations required in the bucket classification step in a fixed-size MSM can be predicted within a small range. Therefore, reducing the bubbles in the PADD pipeline leads to the depreciation of computation time.

In our proposed architecture, window size directly impacts the number of buckets, thereby affecting both area and cost. Moreover, it plays a crucial role in determining the implementation speed of MSM. Therefore, the selection of the window size cannot be arbitrary. In Section 4.1, we introduce ATP, which comprehensively considers both area cost and performance, to determine the recommended window size for our architecture. This is a reference for optimizing the balance between area cost and performance.

3.4 ωNAF and Precomputation

Within the bucket method, the number of buckets is contingent upon the window size c. It is essential to note that a larger c brings about a proportional increase in the area overhead, necessitated by the storage of $2^c - 1$ points. Our objective is to diminish the number of buckets without compromising the window size, with the ultimate aim of reducing the associated area overhead.

Points on an elliptic curve possess a distinctive property: in affine coordinates, transforming a point G to $-G$ is a cost-effective operation that only requires negating the X coordinate to obtain $-X$. We can leverage this property to obtain $-G_i$ from G_i easily. Furthermore, the unsigned scalar with b-bit can be converted to a group of signed representations with c-bit as follows: $unsigned\ (a_{i(m-1)}, \ldots, a_{i1}, a_{i0}) \rightarrow (signed\ a_{i(m-1)}, \ldots, signed\ a_{i1}, signed\ a_{i0})$. This step serves as a preprocessing stage and can be accomplished during scalar generation. This property has been leveraged in [2, 35] to enhance the area efficiency. Building upon this approach, we extend the optimization to attain superior area performance. In [36], two points are supplied per cycle. In contrast, our architecture operates with a requirement of only one point per cycle. Nevertheless, we hope to maximize the utilization of the available bandwidth. To achieve this, we propose incorporating the double version of point G_i, denoted as $2G_i$, in each cycle. When $2G_i$ is selected, the corresponding index a_{ij} is halved. In simple terms, $a_{ij} \times G_i$ is equivalent to $\frac{a_{ij}}{2} \times 2G_i$.

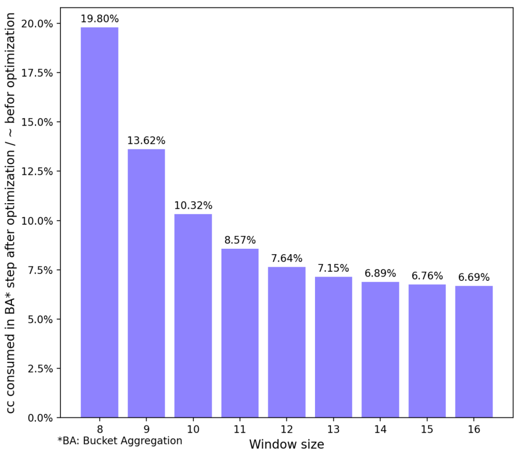


Fig. 7. *Optimization rate.* A lower value is better.

Through the synergistic integration of the ωNAF method [19] and the precomputation technique [18], a notable reduction in the required number of buckets can be achieved. In Figure 6, a straightforward example is presented to illustrate the combined application of these two techniques in compressing scalars. Whenever PriorMSM is provided with G_i and $2G_i$, the selection between the two points is determined by the lowest bit of a_{ij}. Additionally, the sign bit, which is the highest bit of a_{ij}, is examined. When it is 1 (indicating a negative value), PriorMSM discards it and utilizes the remaining bits of a_{ij}. Simultaneously, the negative version of the point is selected. As a result, the scalar range contracts from $[1, 2^c]$ to $[1, 2^{c-1}]$, leading to a 50% reduction in the number of required buckets. Subsequently, the lowest bit of a_{ij} is re-evaluated; if the reduced scalar is even, the lowest bit of a_{ij} is discarded. This operation effectively compresses all even scalars in the range $(1, 2^{c-1}]$ to the range $[1, 2^{c-2}]$, resulting in a 25% reduction in the required number of buckets. By combining these two approaches, only $(1 - \frac{1}{2}) \cdot (1 - \frac{1}{4}) \cdot 2^c$ buckets are needed, leading to a substantial 62.5% saving in on-chip storage usage.

4 Evaluation

4.1 Design Sensitive Analysis

We evaluate the effectiveness of PriorMSM across a range of degrees, spanning from 2^{20} to 2^{26}, and with varying window sizes ranging from 8 bits to 16 bits. We examine two key factors that could impact the efficiency of our proposed architecture. The initial factor under consideration is the partitioning algorithm discussed in Section 3.2. The fully pipelined PADD unit opens avenues for harnessing the parallelism inherent in the bucket aggregation step. This partitioning algorithm significantly reduces the time required for the bucket aggregation step, a process solely dependent on the window size. In our design, we strategically partition the Buckets into 16 groups to facilitate parallel processing, taking into account both efficiency and design complexity. In Figure 7, the *optimization rate* statistics for different windows are presented, highlighting that our partitioning algorithm leads to substantial improvements in a single bucket aggregation step.

The magnitude of improvement corresponds directly to the window size, with notable gains observed as the size increases. Specifically, when the window size is set to 16 bits, the *optimization rate* is 6.69%. This means that the optimized bucket aggregation step consumes only about 6.69% of the clock cycles compared to the unoptimized version. The figure also implies the presence of an upper limit to the improvement effect, potentially dictated by the number of partitions. While adjusting

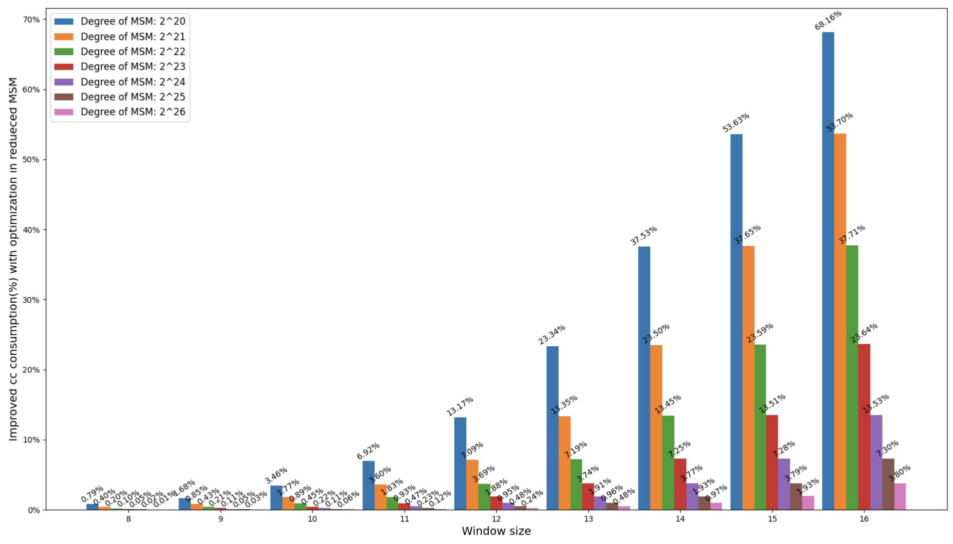


Fig. 8. Proportion of saved clock cycles with our optimization method in MSM.

the number of partitions might yield even more favorable results, it is essential to acknowledge that such optimization could introduce greater complexity to the hardware implementation. This aspect remains a topic for future exploration and consideration.

While the enhancement of the partitioning algorithm significantly improves the bucket aggregation step, our primary emphasis is on optimizing the reduced MSM, encompassing the bucket classification, bucket aggregation, and result aggregation steps. Figure 8 illustrates the percentage of clock cycles saved during the MSM calculation for different window sizes, showcasing the impact of our optimization method. The enhancements in MSM are notably more pronounced for larger window sizes, as the clock cycle consumption in the bucket aggregation step is already comparatively low for smaller window sizes. Nonetheless, the smaller disparity between the degree of MSM and window size can lead to a non-negligible clock cycle consumption in the bucket aggregation step, emerging as a potential bottleneck for MSM. As an illustration, when the degree is 2^{20} and the window size is 16, the partitioning algorithm saves 68.18% of the clock cycle consumption. Remarkably, even in the case of MSM with a degree of 2^{26}, it still contributes to an approximately 3.80% improvement in clock cycle consumption.

We are also curious about the impact of window size on the number of clock cycles consumed by the MSM. In Figure 9, it is evident that MSM with a larger window size necessitates fewer clock cycles, irrespective of the degree of the MSM. In pursuit of a deeper comprehension of the factors contributing to this observation, we gather data on the average number of clock cycles consumed by the reduced MSM and present it graphically in Figure 10(a). Our findings indicate that this data is predominantly contingent on the degree of the MSM. With an increase in the degree, the average number of clock cycles consumed per point in the reduced MSM tends to converge toward 1. Notably, while MSM with a larger window size results in fewer clock cycles consumed, the window size has minimal impact on this trend. This conclusion aligns with our intuition, as a reduction in buckets necessitates more PADD operations. The overall number of clock cycles is computed as the sum of the clock cycles consumed by m reduced MSMs, reinforcing the observed trend.

As the value of c decreases, m increases, resulting in a higher number of clock cycles consumed by the MSM, and conversely, selecting a higher value of c leads to significantly larger areas,

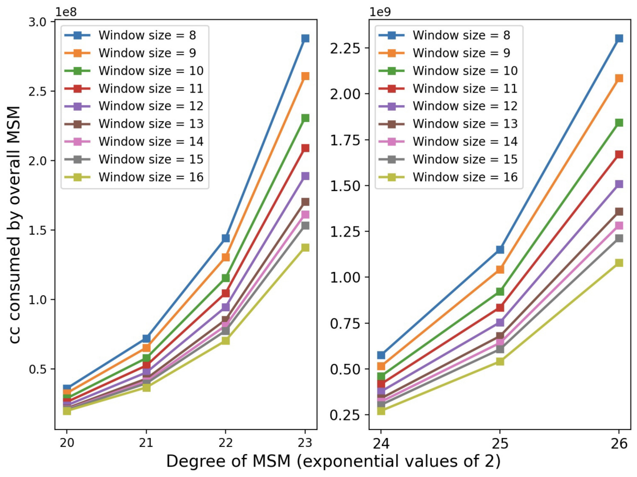


Fig. 9. Total *cc* consumed by overall MSM with different window sizes and degrees ranging from 2^{20} to 2^{26}.

exhibiting exponential growth. Consequently, achieving an optimal tradeoff between area and performance necessitates the selection of a suitable value for c.

Moreover, when examining the execution time and area of our proposed design across different values of c ranging from 8 to 16, we depict the ATP for various window size selections in Figure 10(b). Smaller ATP values are preferred. Figure 10(b) indicates that, in general, a window size of 12 achieves the optimal balance between performance and area (smallest ATP). As the window size increases, ATP sharply rises. This is due to the exponential growth of the bucket's area when the window size increases. However, the reduction in execution time diminishes at a decreasing rate. When the window size becomes sufficiently large, the corresponding bucket's area surpasses that of other units by a significant margin. Essentially, the limiting factor for improving MSM performance lies in the inadequate computational power of the PADD module. Increasing the number of PADD modules would address this, but it introduces larger area requirements and more pipeline bubbles. Moreover, it leads to a linear increase in bandwidth demands.

4.2 Evaluation of Our Implementation

PriorMSM is implemented in Verilog HDL. We synthesize PriorMSM using the Synopsys Design Compiler in the TSMC 28 nm process to assess its performance. The hardware is evaluated based on the BLS12-377 curve, with a scalar of 253 bits, eight Bucket banks, and a point represented in extended coordinates with 4*377 bits. We choose a window size of 12, which is determined based on the analysis of ATP in Section 4.1. This value represents our optimal choice after considering the tradeoff between performance and cost.

Synthesis reports indicate that our MSM architecture can run at 1 GHz in the TSMC 28 nm process. More details about the area and power consumption are shown in Table 5. It can be seen that the majority of the area and power consumption are attributed to the PADD and Buckets. Therefore, devising an efficient PADD, combined with implementing various strategies to diminish the number of buckets, signifies a promising avenue toward constructing a high-performance architecture.

For functionality testing, we use Synopsys VCS and Verdi tools to conduct simulation tests on PriorMSM and various exploratory experiments, to calculate their specific cycle counts and corresponding computational times. We perform multiple simulation tests with different datasets to

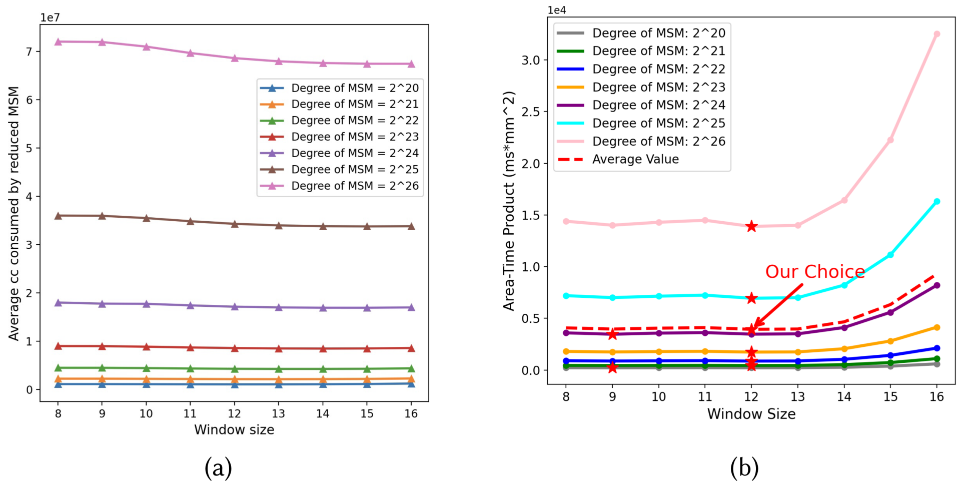


Fig. 10. (a) Average *cc* consumed by reduced MSM with different window sizes and degrees ranging from 2^{20} to 2^{26}. (b) Area-Time Product (ATP) with MSM with a degree from 2^{20} to 2^{26} in different window sizes. The smaller the ATP, the better. The star symbol represents the minimum ATP value and its corresponding window size. The dashed line represents the average ATP within this range of degrees.

Table 5. Area and Power Consumptions of PriorMSM

Module	Submodule	Area (mm^2)	Power (W)
MSM	PADD	5.22 (56.7%)	3.81 (69.8%)
	mul	0.540	0.21
	Buckets	3.44 (37.3%)	1.16 (21.2%)
	FIFOs	0.294 (3.2%)	0.26 (4.8%)
	Others	0.256 (2.8%)	0.23 (4.2%)
Overall		9.21	5.46

ensure the data's reliability and robustness against extreme scenarios. The datasets comprise points from the BLS12-377 curve, commonly utilized in real-world ZKP applications. Scalars are randomly generated based on their corresponding bit lengths. These randomly generated scalars exhibit characteristics of random distribution and unpredictability, aligning with practical scenarios.

We evaluate the performance of PriorMSM (single PADD unit) with degrees ranging from 2^{16} to 2^{26} and compare it with the previous works as shown in Table 6. In actual runtime, scalars and points are stored in external DRAM and sequentially read into the PriorMSM through a small-capacity cache. DRAM effortlessly accommodates the storage of this data; when the degree is 2^{26}, the total size of points and scalars is 20 GB. DRAM can also satisfy the required bandwidth for our operations. In Table 6, the "-" symbol indicates that there is no comparative data available for that particular degree range. Benchmark in PipeZK [36] is based on curve BLS12-381, and with slightly different from BLS12-377. Due to the implementation of a fully pipelined PADD, we largely disregard the impact of these differences on the runtime evaluation of MSM, as they only affect the design of the PADD unit. The speedup in Table 6 is relative to the best performance of

Table 6. Comparison of the Execution Time (ms) of PriorMSM with Different Degrees with Previous Works

Implementation	Platform	Power	Area/ Resources	Degree										
				2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
PipeZK[1] [36]	UMC 28nm 300 MHz	2.38 W	16.86 mm^2	22	46	92	184	368	–	–	–	–	–	–
cuZK [26]	Nvidia V100 –	–	–	–	–	–	27	47	90	171	312	–	–	–
GZKP [27]	Nvidia V100 –	–	–	7	–	20	–	62	–	240	–	1,100	–	4,000
Hardcaml [3]	Xilinx VU9P 278 MHz	–	387k LUTs 733k REGs 2,999 DSPs 883.5 RAMs	–	–	–	499	540	620	780	1,094	–	–	–
PipeMSM [35]	Xilinx U55C 125 MHz	34.9 W	–	17.6	35.9	68.8	136.6	273.0	–	–	–	–	–	–
CycloneMSM [2]	AWS F1 250 MHz	43.47 W	526k LUTs 661k REGs 2277 DSPs 623 RAMs	–	–	–	–	–	–	817.9	1,133	1,761	3,016	5,656
BSTMSM [37]	Xilinx U250 300 MHz	–	410k LUTs 744k REGs 2,920 DSPs 623 RAMs	–	–	23	40	75	145	285	564	1,124	2,242	4,479
PriorMSM[2]	TSMC 28 nm 1 GHz	5.26 W	9.21 mm^2	1.8 (**3.9×**)	3.3 (**10.9×**)	6.2 (3.2×)	12 (2.3×)	24 (2.0×)	47 (1.9×)	95 (1.8×)	189 (1.7×)	377 (2.9×)	754 (3.0×)	1509 (2.7×)

[1]PipeZK utilizes two PEs in MSM. To estimate the performance of PipeZK with a single PE configuration, we simply double the latency, while halving the area and power consumption.
[2] The speedup is relative to the best performance of previous works with the same degree.

previous works with the same degree. The results indicate that our design can achieve a significant improvement in latency.

We separately discuss its comparison with GPU implementations and its comparison with custom hardware implementations. Compared to the GPU implementations [26, 27], PriorMSM achieves a minimum speedup of 1.7×. At MSM with degree 2^{26}, it can achieve a maximum speedup of 3.9×. Compared to the custom hardware implementations [2, 3, 35–37], PriorMSM can achieve a maximum speedup of 10.9×, particularly at MSM with degree 2^{17}. Thanks to the pipeline design of PriorMSM, the proposed architecture can achieve a running frequency of 1 GHz in the TSMC 28 nm process. This frequency surpasses the operating frequency of the previous custom hardware implementations. Additionally, these custom hardware implementations do not have uniform settings for window size (as shown in Figure 9, where different window sizes profoundly impact the latency of MSM). To ensure a fair comparison, we normalize the latency of each work based on the window size and frequency. We use the normalized clock cycle count as a common reference for comparison. As shown in Table 7, we present the relative speedup of our proposed design compared to other works, assuming a speedup of 1 for PriorMSM. A lower value indicates better performance. Due to the larger area and greater computational and storage resources of GPUs, as well as their dynamically high frequencies, it is not feasible to normalize and obtain clock cycle counts of GPU-based works. Furthermore, the design principles behind GPU-based works differ from those of custom circuit-based designs. Therefore, in Table 7, we primarily compare PriorMSM with other works based on FPGA and ASIC to explore the advantages of our architecture relative to baseline designs.

Table 7. Normalized Comparison of Clock Cycle Counts for PriorMSM and Previous Works with Various Degrees[1]

Implementation[2]	Degree										
	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
PipeZK [36]	1.22×	1.39×	1.48×	1.53×	1.53×	–	–	–	–	–	–
Hardcaml [3]	–	–	–	12.5×	6.75×	3.96×	2.46×	1.74×	–	–	–
PipeMSM [35]	1.22×	1.36×	1.39×	1.42×	1.42×	–	–	–	–	–	–
CycloneMSM [2]	–	–	–	–	–	–	2.87×	2.00×	1.56×	1.33×	1.25×
BSTMSM [37]	–	–	1.21×	1.08×	1.02×	1.00×	0.98×	0.97×	0.97×	0.97×	0.96×
PriorMSM	1×	1×	1×	1×	1×	1×	1×	1×	1×	1×	1×

[1]Speedup data are based on normalized window size and frequency in terms of the number of clock cycles.
[2]In PriorMSM, we set $c = 12$, when CycloneMSM sets $c = 16$, PipeMSM sets $c = 12$, PipeZK sets $c = 4$, and Hardcaml and BSTMSM set $c = 13$.

From Table 7, it can be observed that even without considering the impact of operating frequency and window size, our proposed design still achieves an improvement in clock cycle counts. We have overcome many shortcomings present in previous works, resulting in at least a 22% improvement in clock cycle counts compared to previous custom hardware-based works except BSTMSM [37]. This enhancement stems from the advantages of our proposed architecture, which includes multi-FIFO and multi-bank architectures based on PBSM, efficient bucket aggregation step improvements, and efficient PADD unit implementations. We believe that the point pairing and scheduling mechanisms of PipeZK [36] and PipeMSM [35] are rather conservative, and the selection of window size in PipeZK is also conservative, with insufficient hardware-friendly designs for buckets. This leads to excessive data backpressure and pipeline bubbles, which could be further improved by increasing the point pairing rate. Our proposed multi-FIFO and multi-bank architecture aims to maximize the point pairing rate and accommodate more pending point pairs, thus achieving higher overall pipeline efficiency and ultimately yielding significant performance improvements. CycloneMSM [2] and Hardcaml [3] employ a mixadd-based PADD design, which introduces data dependencies during point pairing. This limitation arises because points output from PADD can only be added with external input points, rather than supporting point addition between two points output from PADD. In other words, this approach also reduces the success rate of pairings. In contrast, our design employs a multi-FIFO and multi-bank architecture based on PBSM, facilitating the addition of all types of points within our architecture and minimizing unnecessary pipeline stalls, thereby achieving higher utilization of PADD units and further enhancing the implementation speed of MSM. It should be noted that our proposed MSM hardware architecture supports the bucket classification step, bucket aggregation step, and result aggregation step. In contrast, PipeZK [36], CycloneMSM [2], and Hardcaml [3] only support the former. Hence, our work provides a more comprehensive implementation for MSM.

BSTMSM has implemented all three steps of the Pippenger algorithm and achieves a maximum 4% faster implementation speed compared to our proposed design. We believe this is because BSTMSM utilizes true dual-port SRAM as the Buckets, which has a larger area compared to the simple dual-port SRAM we use. Additionally, BSTMSM employs an SRAM capacity equivalent to $\frac{4}{3}m$ times our reduced MSMs (where $m = 22$ when the window size is 12), resulting in an effective area exceeding 100 mm^2 (as referenced in Table 5). Despite this, more buckets enable them to store a substantial amount of intermediate data and efficiently process it, leveraging the pipeline characteristics of the PADD unit. However, this results in the buckets occupying a significant proportion

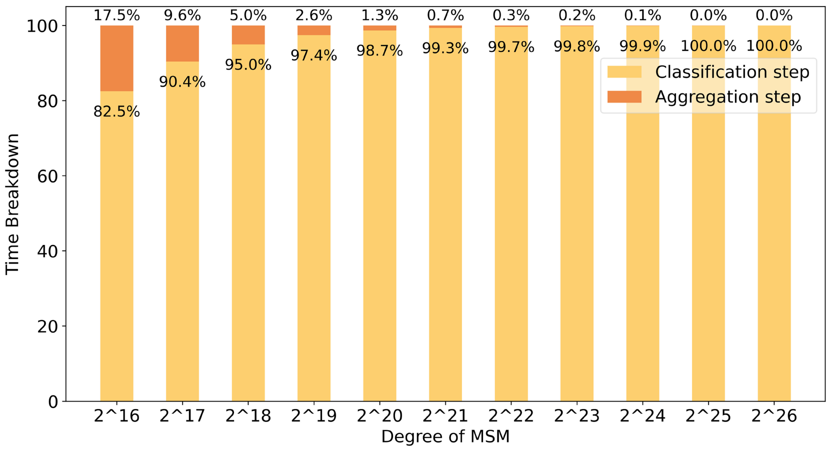


Fig. 11. Time breakdown of the classification and aggregation steps of MSM with degrees form 2^{16} to 2^{26}. The aggregation step includes bucket aggregation and result aggregation. 0.0% represents a negligible portion of time.

of the chip area. From an ATP perspective, this approach is highly impractical. The substantial increase in area only leads to a marginal, even negligible, reduction in latency. Moreover, the larger bucket's area also introduces higher power consumption and presents challenges in layout and routing, thereby impacting performance. In contrast, PriorMSM achieves the same functionality with 288 KB of cache while maintaining nearly consistent speeds. In Figure 11, we present the time breakdown for the classification step and aggregation step of PriorMSM with degrees from 2^{16} to 2^{26}. It can be observed that the time for MSM is primarily determined by the classification step, and as the degree of MSM increases, the time for the aggregation step becomes almost negligible. This is because the clock cycle count for the classification step is largely determined by the degree of MSM, while the clock cycle consumption for the aggregation step is determined solely by the window size. With the window size fixed, the clock cycle count for the aggregation step remains constant. When the degree of MSM is relatively low, the aggregation step can become a bottleneck for the implementation of MSM. We optimize the aggregation step using the pipelining characteristics of PADD, reducing its time proportion with only a limited bucket capacity. As a result, our optimization eliminates the constraint of the aggregation step on the implementation time of MSM. Here, we set the window size to 12. If the degree of MSM is smaller or larger, it might be necessary to reconsider the value of window size. However, the MSM degree we select is within the range of commonly used parameters.

Similarly, we also analyze the reasons for not using single-port SRAM as Buckets. While single-port SRAM with the same capacity incurs less area overhead, it significantly hampers performance as it only supports read or write operations within the same cycle. We theoretically analyze this scenario: if both the new input point and output point from PADD are valid and belong to the same bank (with a theoretical probability of $\frac{1}{8}$, given the eight banks), concurrent read and write operations are no longer supported (with a theoretical probability of $\frac{1}{2}$). As a result, more points are frequently directed to FIFO_U3 to elevate its priority. Theoretically, this would result in a $\frac{1}{16}$ increase in latency; however, we anticipate even longer latency in practice. This scenario reduces the success rate of pairings, impacting PADD utilization and introducing more pipeline bubbles. These bubbles propagate back to the Scheduler, further diminishing the success rate of pairings.

We also conduct a brief statistical analysis on the power consumption and resource utilization of different designs, which we present in Table 6. The analysis of power consumption and resource utilization here does not apply to GPU-based designs because GPUs are power-intensive devices with abundant resources. Unknown data is represented with a dash, "-". MSM designs are characterized by high resource consumption. Even when using high-end Xilinx FPGAs, implementing MSMs on them is not an easy task due to the limitations imposed by the quantity of DSPs and SRAM available on the FPGA. Integrating multiple MSMs on a single FPGA to enhance performance is quite challenging. We also need to consider resource redundancy to achieve better layout and routing, which translates to higher operating frequency. This, however, limits the resource utilization of the FPGA. During runtime, many LUTs and FFs remain idle, resulting in wasted silicon area. For instance, in HARDCAML, DSP utilization is 43% and URAM utilization is nearly 50%, but LUTs and register utilization is only 30%. Additionally, their operating frequency is still restricted, and they incur a significant power consumption cost. For instance, PipeMSM still generates a power overhead of 34.9 W when running at 125 MHz. Of course, this is a characteristic inherent to FPGA devices. ASIC designs are more suitable for balancing performance and area because we can adopt a more balanced allocation of cache and computational resources, guided by ATP metrics. In our ASIC-based design, all logic resources are maximally utilized to achieve better area efficiency. We have also implemented optimizations to enhance hardware utilization, such as improving the bucket aggregation algorithm to fully leverage the pipelining features of the PADD unit and the multi-bank bucket structure. In the 28 nm process, our synthesis tool reports an area overhead of 9.21 mm^2 and a power overhead of 5.26 W. Compared to PipeZK, although we choose a different process technology, considering that our design can operate at 1 GHz, significantly higher than PipeZK's target frequency of 300 MHz, PriorMSM incurs higher power consumption. In terms of area consumption, we believe that PriorMSM features a more optimized PADD design and improved on-chip memory utilization, hence resulting in a smaller area overhead compared to PipeZK.

Furthermore, design sensitivity analysis has guided our parameter selection, a consideration lacking in previous works. Our choice of window size is informed by an analysis of the ATP metric, aiming to strike an optimal balance between performance and area overhead. Similarly, we are also interested in how hardware utilization affects the performance and resource overhead of other works under different parameter configurations. Unfortunately, previous works have not discussed the impact of hardware utilization on the implementation of MSM accelerators. This underscores the importance of our experiments, where design sensitivity analysis aids in a more detailed exploration of the implementation of MSM accelerators in practice.

5 Conclusion

In this article, we present PriorMSM, an efficient hardware architecture designed to accelerate Multi-Scalar Multiplication. The architecture adopts the bucket method, augmented by a Priority-Based Scheduling Mechanism aimed at minimizing pipeline bubbles during MSM computation, thereby optimizing PADD utilization. We propose a novel bucket aggregation algorithm, resulting in a significant reduction in the time required for this crucial step. Additionally, the article delves into strategies for minimizing on-chip SRAM area through the utilization of the ωNAF method and precomputation techniques. We delve into the sensitivity of MSM design to window size. Experimental data suggests that higher window sizes lead to nearly proportional speed improvements, given the optimization of the bucket aggregation step. We use the ATP metric to find the optimal window size for both area and performance in PriorMSM, guiding practical applications and future works. Furthermore, we highlight the performance improvements achieved by PriorMSM using the TSMC 28 nm process and Synopsys Design Compiler tools.

Appendix

A Mathematical Analysis of the Optimized Bucket Method

According to Algorithm 1, we can make a rough estimation of the number of point additions required for MSM, denoted as t. In the bucket classification step, a total of $m(N - 2^c + 1)$ PADD operations are necessary. The bucket aggregation step requires $m(2^{c+1} - 2)$ PADD operations. Finally, during the result aggregation step, $b - c + m - 1$ PADD operations are needed. Hence, t can be determined as follows:

$$t = m(N + 2^c) + b - c. \tag{16}$$

In the bucket classification step, required clock cycles can be roughly estimated as $m(N - 2^c + 1)$, considering that the PADD operations among buckets with different indexes are mutually independent, and the PADD unit is fully pipelined. Nonetheless, the actual number of clock cycles needed is slightly greater than this value, taking into account factors such as pipeline initialization, delays from conflicts, and pipeline stalling. If N is adequately large, the value of t can be approximated by $m(N - 2^c + 1)$, as demonstrated in Section 4. In the bucket aggregation and result aggregation steps, considering the dependencies between intermediate values during computation, the number of clock cycles can be represented as $(m(2^{c+1} - 2) + b - c - 1) \cdot D$.

In terms of the number of clock cycles consumed by the bucket aggregation step, the optimization achieved through the partitioning algorithm in bucket aggregation allows for significant improvements. We can divide all bucket values into H groups, rewriting Equation (11) as Equation (14). The naive method, as referenced in Algorithm 1, can be employed to accomplish bucket aggregation for each group. The total number of PADD operations is $2 \cdot M \cdot H$, which is equivalent to 2^{c+1}. The number of clock cycles consumed in computing the first term of Equation (14) is presented as

$$M \cdot D + 2H. \tag{17}$$

In Equation (17), the aggregation of each distinct group is tightly organized within the pipeline. Therefore, it is necessary to consider only the clock cycles corresponding to the number of groups during entry and exit from the pipeline. This is also the essence of why this optimization algorithm achieves significant results, as we fully utilize the pipeline for the PADD unit.

Subsequently, we utilize intermediate values during the computation of the first term to obtain the second term of Equation (14) by multiplying it with $(h - 1) \cdot M$. We aggregate the second term of Equation (14) together. The estimated time consumption for this step is

$$((c - 2) + \log_2(H - 1)) \cdot D. \tag{18}$$

Finally, we aggregate each first term of the H groups together and add it to the aggregated result of the second term of Equation (14), obtaining the bucket aggregation result. The number of clock cycles consumed is

$$(\log_2 H + 1) \cdot D. \tag{19}$$

Therefore, summarily, with these enhancements, the total number of clock cycles consumed in the optimized bucket aggregation step in the reduced MSM is the sum of Equations (17) through (19), as shown in Equation (20):

$$((c + M - 1) + 2\log_2 H) \cdot D + 2H. \tag{20}$$

On the other hand, the number of clock cycles consumed by the unoptimized bucket aggregation step in the reduced MSM is given by Equation (21):

$$(2^c - 1) \cdot D. \tag{21}$$

We introduce *optimization rate*, calculated as Equation (20) divided by Equation (21), as a measure of the effectiveness of our proposed partitioning optimization algorithm on the bucket aggregation step, as shown in Equation (22):

$$\frac{(c + M - 1) + 2\log_2 H}{2^c - 1} + \frac{2H}{(2^c - 1) \cdot D}. \tag{22}$$

References

[1] Wikipedia contributors. 2024. Montgomery curve. Retrieved November 13, 2023 from https://en.wikipedia.org/wiki/Montgomery_curve.

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

[3] Ben Devlin and Andy Ray. [n. d.]. HARDCAML. Retrieved October 15, 2023 from https://zprize.hardcaml.com/msm-overview.html

[4] Diego F. Aranha, Youssef El Housni, and Aurore Guillevic. 2022. A survey of elliptic curves for proof systems. *Designs, Codes and Cryptography* (2022), 1–46.

[5] Utsav Banerjee and Anantha P. Chandrakasan. 2021. A low-power BLS12-381 pairing cryptoprocessor for Internet-of-Things security applications. *IEEE Solid-State Circuits Letters* 4 (2021), 190–193. https://doi.org/10.1109/LSSC.2021.3124074

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, 781–796.

[7] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. 2008. Twisted Edwards curves. In *Progress in Cryptology: First International Conference on Cryptology in Africa (AFRICACRYPT'08), Proceedings 1*. Springer, 389–405.

[8] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. 2023. *ConsenSys/gnark: v0.8.0*. https://doi.org/10.5281/zenodo.5819104

[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP'18)*. 315–334. https://doi.org/10.1109/SP.2018.00020

[10] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Advances in Cryptology (EUROCRYPT'20)*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768.

[11] Craig Costello and Benjamin Smith. 2017. Montgomery Curves and their Arithmetic: The Case of Large Characteristic Fields. Cryptology ePrint Archive, Paper 2017/212. https://eprint.iacr.org/2017/212

[12] Tanja Lange and Daniel J. Bernstein. [n. d.]. Explicit-Formulas Database. Retrieved November 11, 2023 from https://www.hyperelliptic.org/EFD/

[13] David Gabay, Kemal Akkaya, and Mumin Cebe. 2020. Privacy-preserving authentication scheme for connected electric vehicles using blockchain and zero knowledge proofs. *IEEE Transactions on Vehicular Technology* 69, 6 (2020), 5760–5772. https://doi.org/10.1109/TVT.2020.2977361

[14] Ariel Gabizon, Zachary J. Williamson, and Oana-Madalina Ciobotaru. 2019. PLONK: Permutations over LaGrange-bases for oecumenical noninteractive arguments of knowledge. IACR Cryptology ePrint Archive, Paper 2019/953. https://api.semanticscholar.org/CorpusID:201685538

[15] S. Goldwasser, S. Micali, and C. Rackoff. 1985. The knowledge complexity of interactive proof-systems. (1985), 291–304. https://doi.org/10.1145/22145.22178

[16] G. Gong. 2023. Speeding up Multi-Scalar multiplication over fixed points. Retrieved November 11, 2023 from http://www.fields.utoronto.ca/talks/Speeding-Multi-Scalar-Multiplication-over-Fixed-Points

[17] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 305–326.

[18] G. Gutoski. [n. d.]. Multi-Scalar Multiplication: State of the Art and New Ideas. Retrieved February 22, 2023 from https://www.slideshare.net/GusGutoski/multiscalar-multiplication-state-of-the-art-and-new-ideas

[19] D. Hankerson, A. J. Menezes, and S. Vanstone. 2006. *Guide to Elliptic Curve Cryptography*. Springer New York. https://books.google.com/books?id=V5oACAAAQBAJ

[20] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. 2008. Twisted Edwards curves revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 326–343.

[21] Youssef El Housni and Gautam Botrel. 2022. EdMSM: Multi-scalar-multiplication for SNARKs and faster montgomery multiplication. Cryptology ePrint Archive, Paper 2022/1400. https://eprint.iacr.org/2022/1400

[22] Haiping Huang, Peng Zhu, Fu Xiao, Xiang Sun, and Qinglong Huang. 2020. A blockchain-based scheme for privacy-preserving and secure sharing of medical data. *Computers & Security* 99 (2020), 102010.

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

[24] Zia U. A. Khan and Mohammed Benaissa. 2016. High-speed and low-latency ECC processor implementation over GF(2^m) on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 1 (2016), 165–176.

[25] Changxu Liu, Hao Zhou, Lan Yang, Jiamin Xu, Patrick Dai, and Fan Yang. 2024. Gypsophila: A scalable and bandwidth-optimized multi-scalar multiplication architecture. In *2024 61st ACM/IEEE Design Automation Conference (DAC'24)*.

[26] Tao Lu, Chengkun Wei, Ruijing Yu, Yi Chen, Li Wang, Chaochao Chen, Zeke Wang, and Wenzhi Chen. 2022. cuZK: Accelerating Zero-Knowledge Proof with a Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs. Cryptology ePrint Archive, Paper 2022/1321. https://eprint.iacr.org/2022/1321

[27] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS'23)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/3575693.3575711

[28] Omer Shlomovits. [n. d.]. Introduction to ECNTT from StarkWare Sessions 2023. Retrieved November 12, 2023 from https://medium.com/@ingonyama/intro-to-ecntt-from-starkware-sessions-2023-2aa2cede9fe6

[29] Nicholas Pippenger. 1976. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (SFCS'76)*. 258–263. https://doi.org/10.1109/SFCS.1976.21

[30] Niels Pirotte, Jo Vliegen, Lejla Batina, and Nele Mentens. 2018. Design of a fully balanced ASIC coprocessor implementing complete addition formulas on Weierstrass elliptic curves. In *2018 21st Euromicro Conference on Digital System Design (DSD'18)*. 545–552. https://doi.org/10.1109/DSD.2018.00095

[31] Bahram Rashidi. 2017. A survey on hardware implementations of elliptic curve cryptosystems. *arXiv preprint arXiv:1710.08336* (2017).

[32] Bahram Rashidi. 2018. Efficient hardware implementations of point multiplication for binary Edwards curves. *International Journal of Circuit Theory and Applications* 46, 8 (2018), 1516–1533.

[33] Bahram Rashidi, Sayed Masoud Sayedi, and Reza Rezaeian Farashahi. 2016. An efficient and high-speed VLSI implementation of optimal normal basis multiplication over GF(2m). *Integration* 55 (2016), 138–154. https://doi.org/10.1016/j.vlsi.2016.05.006

[34] Raziyeh Salarifard, Siavash Bayat-Sarmadi, and Hatameh Mosanaei-Boorani. 2018. A low-latency and low-complexity point-multiplication in ECC. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 9 (2018), 2869–2877. https://doi.org/10.1109/TCSI.2018.2801118

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

[36] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. PipeZK: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 416–428. https://doi.org/10.1109/ISCA52012.2021.00040

[37] Baoze Zhao, Wenjin Huang, Tianrui Li, and Yihua Huang. 2023. BSTMSM: A high-performance FPGA-based multi-scalar multiplication hardware accelerator. In *2023 International Conference on Field Programmable Technology (ICFPT'23)*. 35–43. https://doi.org/10.1109/ICFPT59805.2023.00009

[38] Xudong Zhu, Haoqi He, Zhengbang Yang, Yi Deng, Lutan Zhao, and Rui Hou. 2024. Elastic MSM: A fast, elastic and modular preprocessing technique for multi-scalar multiplication algorithm on GPUs. Cryptology ePrint Archive, Paper 2024/057. https://eprint.iacr.org/2024/057

Received 3 March 2024; revised 1 June 2024; accepted 30 June 2024