IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–15. DOI:XXXXXXXX

RcLboMMSM : Load Balancing Optimization of Multi-Scalars Multiplication in Resource-Constrained Environments with Multi-GPU Systems

Jiayong Li, Hao Meng and Weizhe Zhang

Harbin Institute of Technology, Harbin, China, sanrenfengna@gamil.com

Abstract.
In modern cryptography, zero-knowledge proof technology is essential for protecting data privacy and enhancing security. zkSNARK is one of the most practical zero-knowledge proof protocols and has been widely used in many fields such as cryptocurrency, smart contracts, decentralized games, and secure voting systems. However, zkSNARK contains a large number of computationally intensive operations, especially multi-scalar multiplication (MSM) calculations at large scale, which account for more than 70% of the total computing time. At the same time, as the computational tasks required for verification become increasingly complex, the computational requirements of MSM also increase. Processing larger scale and more batches of MSM becomes a key challenge, but the resources of consumer-grade GPUs are limited. How to process large-scale MSM tasks under resource-constrained GPUs becomes a key challenge.
To address these challenges, we propose RcLboMMSM, an MSM computing framework based on the Pippenger algorithm. RcLboMMSM is optimized for sparse scalar loads and can dynamically adjust the balance between performance and memory usage on resource-constrained consumer-grade GPUs. In addition, in a multi-GPU environment, RcLboMMSM can dynamically allocate tasks according to the elliptic curve type, data size, and hardware resources to ensure load balancing, thereby efficiently processing large-scale MSM tasks.
Experimental results show that RcLboMMSM significantly improves the computing efficiency of MSM, solves the resource limitation problem of consumer-grade GPUs, and provides strong support for the practical application of ZKP technology.

Keywords: Pippenger · zkSNARK · Multi-GPU Systems · Multi-scalar Multiplication · Resource-Constrained

1 Introduction

In recent years, zero-knowledge succinct non-interactive arguments of knowledge (zkSNARK) have attracted great interest from theoretical development to practical implementation, as it provides an elegant solution for privacy protection. Popular examples include anonymous transactions in Zcash and smart contract verification of private inputs in Ethereum. In zkSNARK, the verifier can check the proof it receives without interacting with the prover, but is bottlenecked by several high-impact, expensive operations. The most important of these is the multi-scalar multiplication (MSM), which is used to submit large polynomials during the proof process. The MSM task calculates the sum of multiple points on the elliptic curve with different coefficients during the proof generation process.

Depending on the scale of the circuit, the number of points to be calculated can range from 2^{14} to 2^{26} or even higher, accounting for 70-85% of the total running time.

Therefore, using hardware to accelerate the MSM task in zkSNARK is becoming a key research focus in contemporary cryptography and computing. Graphics processing units (GPUs) are able to provide powerful parallel computing capabilities, but their limited memory capacity has become a major bottleneck for processing large-scale and increasingly complex MSM problems. As the computations we need to prove become more complex, MSMs are also becoming more complex. In the 2023 ZKP Acceleration Challenge ZPrize, the challenge faced by participants is to accelerate MSM computations with input sizes up to $N = 2^{26}$, or 67 million inputs.

2 BACKGROUND AND MOTIVATION

2.1 The zkSNARK Protocol

A zkSNARK is a cryptographic primitive that enables a prover to generate a proof π such that both the size of π and the cost of verifying it are sublinear in the size of the verifier. zkSNARKs are typically composed of three algorithms: Setup, Prove, and Verify. The Setup and Prove algorithms involve multiple large computational instances of large-scale polynomial operations in the prime field Fr[x] and multi-scalar multiplications on elliptic curve points. When dealing with high-order polynomials, the best approach for fast arithmetic in Fr[x] is the Fast Fourier Transform (FFT), while the best implementation of large-scale MSMs is the Pippenger algorithm and its variants. In the zkSNARK workflow, the prover generates a proof by performing a series of point-based operations on the elliptic curve group, while the verifier verifies the correctness of the proof through multiple equation pairings. The prover and verifier are typically required to use only linear operations on the points constructed in the public reference string, which are actually MSMs on fixed points. Therefore, all popular zkSNARK implementations, such as Zcash, TurboPLONK, Bellman, and gnark, have chosen Pippenger's bucket method or its variants to accelerate MSM computations on fixed points. Among multiple zkSNARK algorithms, MSM is the most time-consuming computational part and provides key support for data hiding. MSM is often used in the commitment computation phase of algorithms such as Plonk, Groth16, and Halo. However, since MSM is computationally intensive, it takes hundreds of milliseconds or even tens of seconds to complete MSM tasks using the CPU, becoming a performance bottleneck in applications.

2.2 Elliptic curves

Elliptic curve (EC) is a smooth projected algebraic curve composed of EC points. The definition of elliptic curve (Weierstrass form) defined on a finite field is shown in Eq. (1)

$$(y^2 = x^3 + ax + b) \tag{1}$$

The point at infinity, denoted as O, is the identity element in the Abelian group of all elliptic curve (EC) points.

Points on elliptic curves support several operations, including point addition (PADD), point doubling (PDBL), point negation (PNEG), and point scalar multiplication (PMUL). Among these, PADD is the fundamental operation. PDBL is a special case of PADD, where the operation is performed on two identical points. The PNEG operation is relatively inexpensive, requiring only the negation of the y-coordinate of the EC point. Scalar multiplication (PMUL) involves repeated PADD operations. Notably, point addition on elliptic curves is computationally intensive due to the required modular arithmetic and multiplication operations over large finite fields.

A straightforward method to compute scalar multiplication is to perform repeated point additions, which is inefficient. A more efficient algorithm is the double-and-add method. This algorithm performs a series of PDBL and PADD operations to compute scalar multiplication. For example, to compute $21P$, we use the double-and-add method as illustrated in Figure 1. First, express 21 in binary form and initialize the result to the point at infinity O. Then, perform a PDBL operation at each bit position to double the point. If the bit is 1, use PADD to add the point to the result.

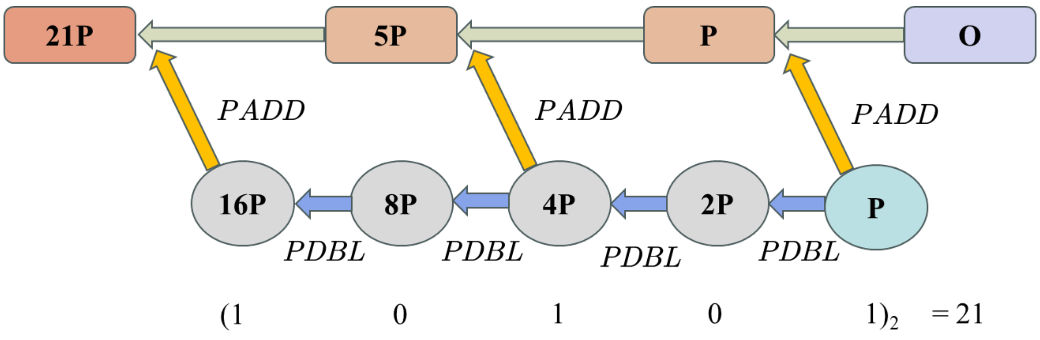


Figure 1: The double-and-add algorithm calculates 21P

Elliptic curve points can be represented using affine coordinates (x, y) or projective coordinates (X, Y, Z). Affine coordinates (x, y) correspond to $(X/Z, Y/Z)$ in projective coordinates. By introducing an auxiliary coordinate $T = XY$, extended projective coordinates (X, Y, Z, T) can be used. Affine coordinates, requiring only two field elements, are efficient for storage but not for point addition due to the expensive modular inversion operation. Projective coordinates avoid this by using (X, Y, Z), and extended projective coordinates (X, Y, Z, T) are even more efficient for point addition.

Most FPGAs implements[] convert points to Edwards curves, which provide fast point addition formulas and more efficient point addition operations. As shown in Eq. 2, the Twisted Edwards curve is preferred for fast calculations due to its efficient addition formula. This efficiency arises from the robustness and completeness of the curve, enhancing its performance in cryptographic protocols.

$$a \cdot x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2 \tag{2}$$

The addition formula of the extended projection coordinates avoids the inefficient modular inversion operation in the Twisted Edwards curve and implements point addition with fewer modular multiplication operations.

Table 1 compares the number of field operations required after applying our transformation. Here, M, S, D, and A represent the number of field multiplications, squares, constant multiplications, and additions or subtractions, respectively.

Table 1: Number of field operations for a point addition comparison

Implementation	Curve	Coordinate	Field Operations
RcLboMMSM	Twisted Edwards	Extended	$8M + 1A$
cuZK	Weierstrass	XYZZ	$12M + 2S$
wlc-msm	Weierstrass	XYZZ	$12M + 2S$
DistMSM	Weierstrass	XYZZ	$12M + 2S$

Compared with previous studies, RcLboMMSM requires only $7M + 8A + 2D$, achieving the least amount of field operations.

2.3 Multi-scalar Multiplication

MSM plays a key role in polynomial commitments in zkSNARK. Given N scalars k_i and N elliptic curve points P_i, the MSM calculation is defined as shown in Eq. (3):

$$Q = \sum_{i=1}^{N} k_i P_i \tag{3}$$

Multi-scalar multiplication requires a large number of PMUL and PADD operations on elliptic curves.Table 2.3 lists the number of bits for points and scalars associated with several elliptic curves commonly employed in ZKP.

Table 2: Number of bits for some elliptic curves

EC	BN254	BLS12-377	BLS12-381	MNT4753
k_i	254 bits	253 bits	255 bits	753 bits
P_i	254 bits	377 bits	381 bits	753 bits

Elliptic curve points are typically composed of large finite field elements ranging from 254 to 753 bits. PMUL and PADD operations must be decomposed into modular multiplication and modular addition operations on these large finite fields, which are computationally intensive. Furthermore, the computational cost of multi-scalar multiplication (MSM) is proportional to the size of the point set, which can reach tens of millions. Consequently, MSM is the most time-consuming operation in zkSNARK.

The Pippenger algorithm can be used to reduce the complexity of MSM calculation. Its core idea is to use the distributive law to eliminate PMUL operations. Especially when the MSM scale is very large, the Pippenger algorithm performs best.

2.4 Pippenger algorithm

Pippenger provides an asymptotically optimal algorithm for MSM. To this day, the Pippenger algorithm and its variants are still the most advanced and widely used algorithms. To calculate $Q = \sum_{i=1}^{N} k_i P_i$, the Pippenger algorithm is described as follows(also see a simple example in Figure 2.4):

1. Task Decomposition: The Pippenger algorithm selects a window size c and converts each λ-bit scalar k_i into $\lceil \lambda/c \rceil$ c-bit scalar slices $k_{i,j}$, satisfying Eq (4):

$$\begin{cases} k_{i,j} = k_i[j \cdot c : j \cdot c + c - 1] \\ k_i = \displaystyle\sum_{j=0}^{\lceil \lambda/c \rceil - 1} k_{i,j} 2^{jc} \end{cases} \tag{4}$$

Therefore, MSM is decomposed into $\lceil \lambda/c \rceil$ window subtask calculate, satisfying Eq (5):

$$\begin{aligned} Q &= \sum_{i=1}^{N} k_i P_i = \sum_{i=1}^{N} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i \\ &= \sum_{j=0}^{\lambda_c} 2^{jc} \sum_{i=1}^{N} (k_{i,j} P_i) \\ &= \sum_{j=0}^{\lambda_c} 2^{jc} Q_j \end{aligned} \tag{5}$$

2. Calculate the Window Subtask Q_j: To calculate the subtask result Q_j, Pippenger performs the following two stages:

- **Bucket Accumulation Stage**: Pippenger introduces a cache point called "bucket". By placing the elliptic curve point P_i with the same scalar value $k_{i,j}$ into a specific bucket with index $k_{i,j}$ ($k_{i,j}$ has c bits, and the bucket corresponding to scalar 0 will not affect the calculation of MSM, so $2^c - 1$ buckets need to be allocated). Add all the points in the same bucket (PADD) to obtain the bucket cumulative value $B_t^{(j)}$, $t \in [1, 2^c - 1]$.

- **Bucket Reduction Stage**: The subtask result Q_j is obtained by taking the weighted sum of all buckets according to their bucket index, as shown in Eq (6):

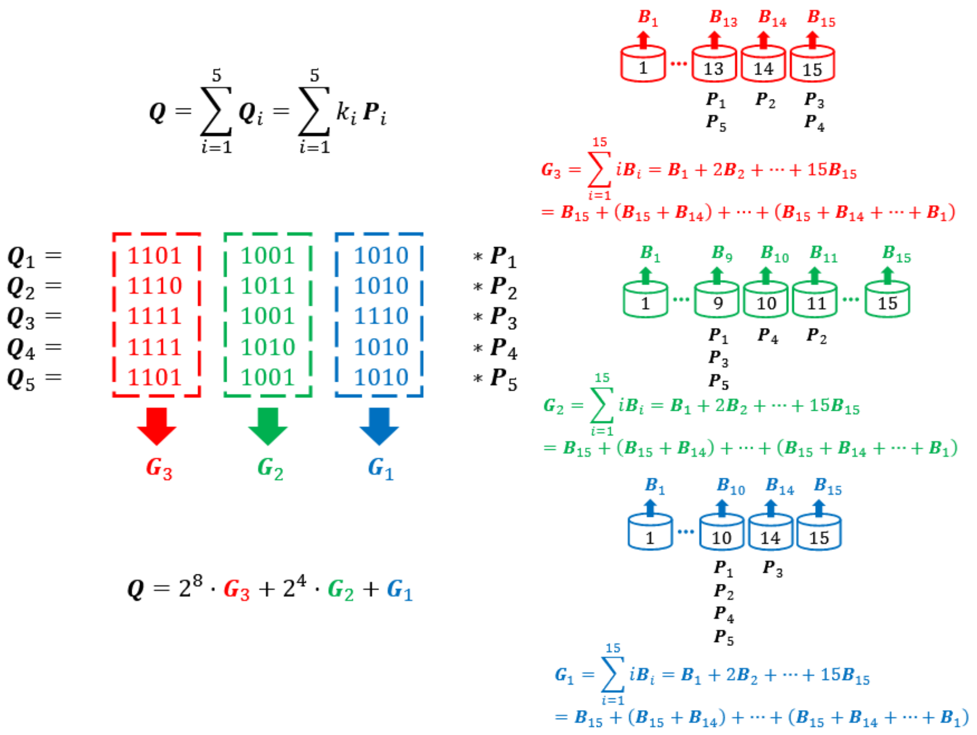
$$Q_j = \sum_{i=1}^{N} k_{i,j} P_i = \sum_{t=1}^{2^c-1} t B_t \tag{6}$$

3. Window Reduction: Calculate MSM based on the subtask results. After obtaining the subtask results, the Pippenger algorithm calculates $Q = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j$ to obtain the MSM result.

2.5 Related work

ZKP has been successfully applied to electronic voting, verifiable database outsourcing, verifiable machine learning, and verifiable outsourcing. Among ZKP algorithms, zk-SNARK is a practical choice and has been successfully deployed in real-world systems such as ZCash and Pinocchio coin. zk-SNARK provides this dual functionality, ensuring privacy and verifiability for such applications.

To improve the performance of zk-SNARK proof generation, previous work has focused on implementing high-performance accelerators using GPUs, ASICs, and FPGAs. These accelerators provide efficient solutions for different hardware platforms. PipeZK is an ASIC-based zk-SNARK hardware accelerator that supports multiple curves. It adopts the 4-step algorithm and the Pippenger algorithm in its POLY and MSM accelerators to achieve excellent performance. PipeMSM is an FPGA-based MSM hardware accelerator that introduces innovative PADD design ideas and a high-performance parallel Barrel Aggregation algorithm. CycloneMSM and CycloneNTT are FPGA-based MSM and POLY accelerators developed by the same team. CycloneMSM converts the PADD operation of BLS12-377 into a Twisted Edwards curve, reducing the use of modular multipliers. HARDCAML is also an FPGA-based MSM hardware accelerator that won the 2022 ZPrize championship. It is the best performing FPGA-based MSM accelerator published to date. CuZK and GZKP are high-performance MSM accelerators based on GPUs. CuZK supports parallel execution on multiple GPUs, while GZKP focuses on single GPU performance.



$Q = \sum_{i=1}^{5} Q_i = \sum_{i=1}^{5} k_i P_i$
$Q_1 =$ 1101 1001 1010 $* P_1$
$Q_2 =$ 1110 1011 1010 $* P_2$
$Q_3 =$ 1111 1001 1110 $* P_3$
$Q_4 =$ 1111 1010 1010 $* P_4$
$Q_5 =$ 1101 1001 1010 $* P_5$
G_3 G_2 G_1
$Q = 2^8 \cdot G_3 + 2^4 \cdot G_2 + G_1$
B_1 B_{13} B_{14} B_{15}
1 13 14 15
P_1 P_5 P_2 P_3 P_4
$G_3 = \sum_{i=1}^{15} iB_i = B_1 + 2B_2 + \cdots + 15B_{15}$
$= B_{15} + (B_{15} + B_{14}) + \cdots + (B_{15} + B_{14} + \cdots + B_1)$
B_1 B_9 B_{10} B_{11} B_{15}
1 9 10 11 15
P_1 P_3 P_5 P_4 P_2
$G_2 = \sum_{i=1}^{15} iB_i = B_1 + 2B_2 + \cdots + 15B_{15}$
$= B_{15} + (B_{15} + B_{14}) + \cdots + (B_{15} + B_{14} + \cdots + B_1)$
B_1 B_{10} B_{14} B_{15}
1 10 14 15
P_1 P_2 P_4 P_5 P_3
$G_1 = \sum_{i=1}^{15} iB_i = B_1 + 2B_2 + \cdots + 15B_{15}$
$= B_{15} + (B_{15} + B_{14}) + \cdots + (B_{15} + B_{14} + \cdots + B_1)$

Figure 2: An example of Pippenger algorithm with 5 MSM scale, 12-bit scalars, and window size 4.

3 RcLboMMSM Design

This section introduces the design of the RcLboMMSM, which optimizes MSM for resource-constrained devices. Compared to previous implementations,RcLboMMSM achieves better performance while significantly reducing resource usage, supporting larger-scale MSM computations. Furthermore, the framework supports not only single GPU but also multi-GPU environments. It dynamically allocates tasks based on different hardware resources to ensure load balancing, thereby efficiently processing large-scale MSM tasks.

3.1 RcLboMMSM Design for Single GPU

This section introduces the design of the RcLboMMSM framework for single GPU, especially consumer-grade GPUs, which optimizes MSM for resource-constrained devices. The details are described as follows, as shown in Algorithm 1.

3.1.1 Optimal Window Size Selection

The window size c is a critical parameter that affects both the parallelism of the MSM module and the distribution of bucket workloads, thereby significantly influencing overall computational efficiency. Generally, larger window sizes decrease the total computational cost of the Pippenger algorithm in MSM. However, increasing c also markedly raises the number of bucket reduction tasks, introducing new performance challenges.

Furthermore, various elliptic curve properties, input point set distributions, and device resources may respond differently to varying window sizes, and their impact on the total

Algorithm 1 RcLboMMSM Single GPU Pseudocode Algorithm

```
Require: Scalar vector $\overrightarrow{k_N} = [k_1, k_2, \ldots, k_N]$, each $k_i$ is a $\lambda$-bit scalar; point vector $\overrightarrow{P_N} = [\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_N]$
Ensure: Compute $\mathbf{Q} = \sum_{i=1}^{N} k_i \mathbf{P}_i$
1: Transfer $\overrightarrow{P}_N$ to GPU and convert to Twisted Edwards extend coordinates system
2: Precompute points, store them in GPU memory, and allocate the necessary GPU resources.
3: Select optimal window size $c$ and compression coefficient M based on the curve type, input size, and GPU resources
4: Convert scalars $\overrightarrow{k_n}$ to signed form
5: for each window $j = 0$ to $\lceil \lambda / c \rceil - 1$ do
6:     Sort scalar slices $k_{i,j}$ and corresponding points $\mathbf{P}_i$ by $k_{i,j}$
7:     Accumulate points into buckets using shared memory
8:     Perform bucket reduction to compute local sums
9: end for
10: Perform inter-window reduction on GPU to obtain partial results
11: Transfer partial results to CPU and complete final reduction to compute $\mathbf{Q}$
12: return $\mathbf{Q}$
```

computation time cannot be overlooked. Therefore, thoughtfully selecting the window size is essential to optimize performance, taking into account specific curve characteristics, input point set properties, and available hardware resources.

To determine the optimal window size c, we employ an offline search strategy. Initially, we conduct preliminary experiments to gather performance data for various window sizes across different data scales and GPU devices. By executing benchmark MSM tasks with varying window sizes c on the GPU, we record performance metrics such as execution time and memory usage, thereby creating a comprehensive performance map. Recognizing that in practical ZKP applications, GPUs are not exclusively dedicated to MSM acceleration, the actual memory constraints may be lower than the GPU's total device memory. Leveraging this performance map, we select the window size that offers the best trade-off between computational efficiency and memory usage, ensuring it adheres to the memory limitations of the target GPU. The detailed search algorithm is outlined in Algorithm 2.

Algorithm 2 Optimal Window Size Selection

```
Require: Data scale, GPU specifications, memory constraints, experimental map
Ensure: Optimal window size $c^*$
1: Initialize a list of candidate window sizes $C = \{c_1, c_2, \ldots, c_n\}$
2: for each $c \in C$ do
3:     if Memory usage with window size $c$ exceeds constraints then
4:         Skip window size $c$
5:     end if
6: end for
7: if No window sizes satisfy memory constraints then
8:     Return an error or default to the smallest window size
9: else
10:     Select the window size $c^*$ from the experimental map that offers the best trade-off between performance and resource utilization
11: end if
12: return $c^*$
```

3.1.2 Elastic Precomputation

The precomputation method reduces running time by generating a series of elliptic curve (EC) points in advance and treating them as additional base points in the MSM. However, this approach increases the number of points by several times compared to the original base points. Consequently, storing these points in GPU memory during runtime results in significant storage overhead, posing a challenge for large-scale MSM computations.

To better balance time and space overhead, this paper adopts the elastic precomputation scheme. Elastic precomputation offers a trade-off between the runtime of the Pippenger algorithm and the additional GPU storage space required. This flexibility allows for an adjustable amount of preprocessing based on specific needs, optimizing performance while accommodating various storage constraints, especially in resource-constrained environments.

Specifically, this paper sets the precomputation compression coefficient M as a control parameter to manage the time and space trade-off, thereby adjusting the data that needs to be precomputed. The precomputed points to be calculated are shown in Eq. (7).

$$2^{Mc}P_i,\ 2^{(2M)\cdot c}P_i,\ \ldots,\ 2^{\left(\left\lceil \frac{\lceil \lambda/c \rceil}{M} \right\rceil - 1\right)\cdot Mc}P_i \tag{7}$$

By setting the compression coefficient M, the original number of windows can be reduced to M. However, this paper does not utilize this technique to reduce bucket space, as we aim to prevent different threads from accessing the same bucket and causing conflicts. In the bucket accumulation phase, we continue to use the original subtask division and add the corresponding flattened weighted precomputed points to the window buckets. After completing bucket accumulation, we allocate threads to perform cross-window bucket reduction, reducing the number of windows to M, and then complete the intra-window bucket reduction and CPU-side window reduction on these M windows.

3.1.3 Scalar conversion and Twisted Edwards Coordinates

Scalar conversion: In the Pippenger algorithm, re-encoding the scalar into a signed binary form in advance can effectively reduce the number of buckets by half, reducing memory overhead while also reducing the time overhead of bucket reduction.

Specifically, this paper converts the scalar k_i from an unsigned c-bit binary representation to a signed representation of each scalar slice in the range of $\left[-2^{c-1}, 2^{c-1}\right]$, as shown in Eq. (8). The specific details of the conversion algorithm are shown in Algorithm 3.1.3.

$$k_i = \sum_{j=0}^{\lambda} k_{i,j} = \sum_{j=0}^{\lambda} \bar{k}_{i,j} \cdot (-1)^{s_{i,j}} \tag{8}$$

Algorithm 3 Scalar Transformation Algorithm

```
Require: Bit-length $\lambda$, window size $c$, $N$ scalars $\{k_i\}_{i\in[N]}$
Ensure: Tuple $\{\widetilde{k}_{i,j}, s_{i,j}\}$
1: $s_{i,-1} \leftarrow 0$, $t \leftarrow 0$
2: for $j = 0$ to $\lceil \lambda/c \rceil$ do
3:     $k_{i,j} \leftarrow k_i[jc : jc + c - 1]$
4:     $t \leftarrow k_{i,j} + s_{i,j-1}$
5:     $s_{i,j} \leftarrow$ if $t > 2^{c-1}$ then $1$ else $0$
6:     $\widetilde{k}_{i,j} \leftarrow$ if $t > 2^{c-1}$ then $t - 2^c$ else $t$
7: end for
```

Twisted Edwards Coordinates: For specific ZKP applications and curves, curve transformations are precomputed; the primary task is point transformation. Algorithm 4 outlines the conversion to the Twisted Edwards curve. Initially, points are transformed from the Weierstrass form to the Montgomery curve as shown in Eq. (9) (A and B are curve parameters):

$$By^2 = x^3 + Ax^2 + x \tag{9}$$

Subsequently, the Montgomery curve is converted to the Twisted Edwards curve. However, not all points can be mapped this way. Points where $x_{\text{mont}} = -1$ or $y_{\text{mont}} = 0$ on the Twisted Edwards curve are invalid. These points are filtered out on the CPU, and their contributions to the MSM result are computed by the CPU without conversion.

Algorithm 4 Transformation to Twisted Edwards Coordinates

```
Require: Elliptic curve parameters $a, b$ in Weierstrass form.
Ensure: Transformed points on the Twisted Edwards curve.
 1: Compute $s = (\sqrt{3a^2 + a})^{-1}$.
 2: Find root $\alpha$ of $x^3 + ax + b = 0$.
 3: Set $A = 3\alpha s$ and $B = s$.
 4: Compute Twisted Edwards parameters:
        $a' = \frac{A+2}{B}$
        $d = \frac{A-2}{B}$
 5: for each point $(x_{\text{weierstrass}}, y_{\text{weierstrass}})$ do
 6:     Compute Montgomery coordinates:
            $x_{\text{mont}} = s(x_{\text{weierstrass}} - a)$
            $y_{\text{mont}} = s \cdot y_{\text{weierstrass}}$
 7:     Compute Twisted Edwards coordinates:
            $x_{\text{twist}} = \frac{x_{\text{mont}}}{y_{\text{mont}}}$
            $y_{\text{twist}} = \frac{x_{\text{mont}} - 1}{x_{\text{mont}} + 1}$
 8:     if $x_{\text{mont}} = -1$ or $y_{\text{mont}} = 0$ then
 9:         Filter out the point and handle MSM contribution on the CPU.
10:     end if
11:     Store $(x_{\text{twist}}, y_{\text{twist}})$
12: end for
```

3.1.4 Reducing Register Pressure in PADD Operations

Optimizing the Point Addition (PADD) operation for large integers heavily relies on managing register pressure—the demand for available registers by concurrent live variables—which directly impacts the kernel's occupancy rate. In PADD operations, each multiplication involves complex large integer modular arithmetic, necessitating numerous computations, memory accesses, and synchronization steps. To mitigate register pressure, RcLboMMSM, similar to DistMSM, rearranges PADD operations in extended coordinates to minimize the peak number of active large integers at any given time.

As illustrated in Figure 3.1.4, this paper analyzed the PADD operation on the Twisted Edwards curve and proposed an optimal execution order accordingly.

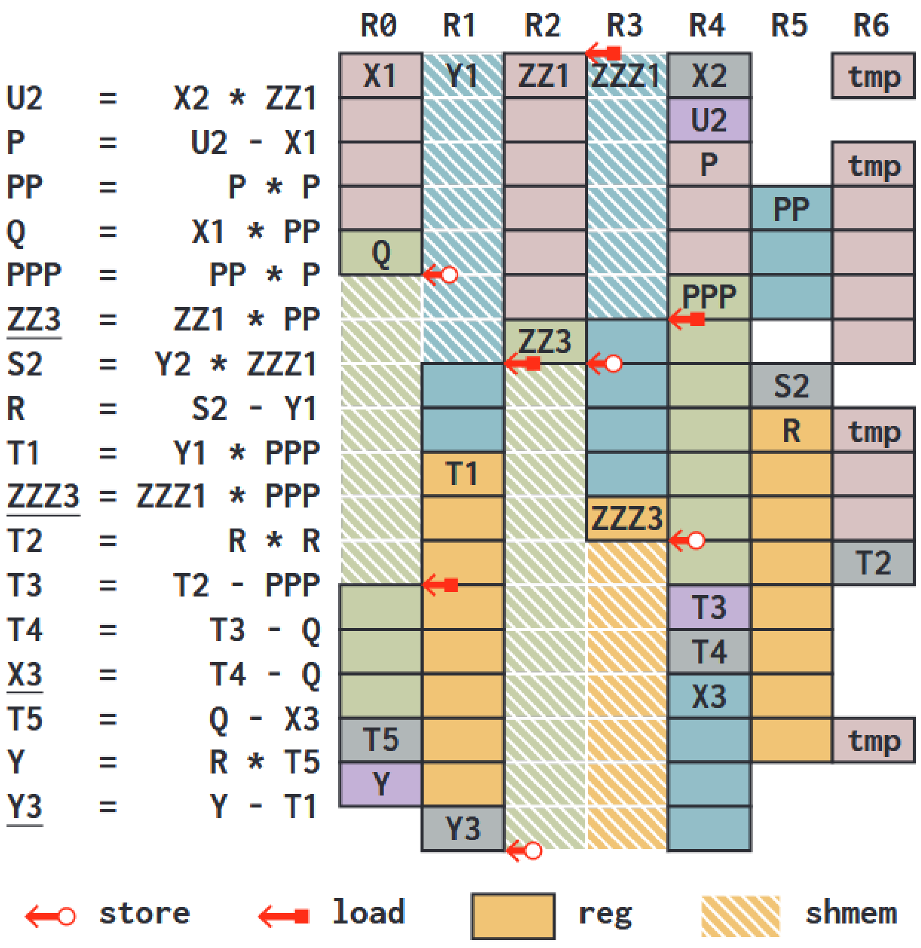


Figure 3: Optimal Execution Order of PADD Operation

3.1.5 Bucket accumulation load balancing optimized for applications

Zero-knowledge proof applications frequently encounter uneven scalar data distributions. In real-world workloads, scalars are highly sparse due to extensive boundary checks and range constraints during system setup, as illustrated in Figure . This prevalence of numerous 0s and 1s results in significant load imbalances in some existing systems.

This paper adopts the load balancing scheme from wlc-msm, which addresses conflict issues by pre-allocating non-conflicting point buffers, caching points within these buffers, and utilizing binary search to assign points to their corresponding buckets.

Specifically, during the bucket accumulation phase, points are written to a pre-allocated non-conflicting static buffer for each thread instead of directly into the buckets. For each of the $\lceil \lambda/c \rceil$ subtasks, a static buffer is allocated, and the starting offset for each thread's writes to the buffer is determined by the provided Eq. 10.

$$\begin{cases} offset_{tid} = tid + \min\{a_i\}_{i\in[s,c)} \\ [s,e) = [[n/N]\cdot tid, [n/N]\cdot(tid+1)) \end{cases} \tag{10}$$

Where a_i is the ith scalar in the sorted subtask. The maximum buffer offset is $N+2^{c-1}$, so the total size of the static buffer that needs to be allocated is $\lceil\lambda/c\rceil\cdot(N+2^{c-1})$. In addition, in order to write the points stored in the buffer to the corresponding buckets later, three auxiliary arrays need to be allocated: buffer_offset is used to store the starting position of the thread writing to the buffer; buffer_index is used to store the bucket offset corresponding to the write point; buffer_use is used to store the buffer size actually used by the thread. The buffer needs to store elliptic curve points. The three auxiliary arrays only need to store numbers. They take up very little space, namely:

$$\begin{cases} \lceil\lambda/c\rceil\cdot N \\ \lceil\lambda/c\rceil\cdot(N+2^{c-1}) \\ \lceil\lambda/c\rceil\cdot N \end{cases} \tag{11}$$

Specifically, in the bucket accumulation phase, we need to complete three stages of operations:

- **Sort the preprocessed sub-scalars and point indices:** For each subtask, in the bucket accumulation phase, the tuple $\{\widetilde{k}_{i,j}, s_{i,j}\}$ needs to be sorted based on the key $\widetilde{k}_{i,j}$. This paper uses the radix sort algorithm provided by the CUDA CUB library.
- **Write to the static buffer:** Accumulate points into the buffer. Before accumulating points into the buffer, the subtask needs to find the precomputed points that need to be added to the bucket instead of the original points. It needs to find the corresponding precomputed points in the precomputed point set based on the subtask number. On this basis, this paper uses the load balancing scheme proposed by wlc-msm, as shown in Algorithm 5. Similar to wlc-msm, this paper uses shared memory to accelerate the read and write of precomputed points, accumulates points into the subtask static buffer, and records the corresponding values in the three auxiliary arrays buffer_offset, buffer_index, and buffer_used to facilitate the next step of accumulating points into the corresponding buckets.
- **Accumulate buffer points into buckets:** Since the subtask has completed the sorting of scalars and point indices, the bucket offsets corresponding to the points stored in the buffer are also sorted in ascending order. Binary search can be used to complete the accumulation task. For each subtask, this paper allocates 2^c-1 threads, where the thread with index tid is used to search for all buffer points corresponding to bucket $(tid+1)$ B and accumulate the buffer points back to the unique bucket (bucket 0 does not contribute to the MSM calculation).

3.1.6 Bucket aggregation optimization

This paper refines the bucket reduction operation of the original Pippenger algorithm into cross-window and intra-window bucket reductions. During preprocessing and bucket accumulation stages, bucket weights are flattened, meaning that every M windows, the corresponding bucket weights are the same. For example, the 0th window can be directly added to the Mth window, compressing the original window into M. In M windows, threads perform bucket reduction on buckets with the same weights in the remaining windows. This way, more points are added to the same bucket, maximizing the advantages of the

Algorithm 5 Using Shared Memory to Accumulate Points to Buffer

```
Require: Sorted tuple pairs $\{(a_i, p_i)\}_{i\in[N]}$, precomputed point set $\{P_i\}_{i\in[N]}$
Ensure: Static buffer buffer, three auxiliary arrays buffer_offset, buffer_index, buffer_used
 1: Obtain boundaries $[s, e)$
 2: pre_bucket_idx = $0xffff$ {Non-existent bucket index}
 3: buffer_offset[tid] = offset = tid + $(a_s >> 1)$
 4: num = 0
 5: smem[2 · tid_inner + 1] = $O$ {tid_inner and $O$ are thread block index and elliptic curve point at infinity}
 6: for $i = s$ to $e - 1$ do
 7:     cur_bucket_idx = $a_s >> 1$
 8:     if cur_bucket_idx ≠ pre_bucket_idx and $i \neq s$ then
 9:         buffer[offset + num] = smem[2 · tid_inner + 1]
10:         buffer_index[offset + num] = pre_bucket_idx
11:         smem[2 · tid_inner + 1] = $O$
12:         num = num + 1
13:     end if
14:     pre_bucket_idx = cur_bucket_idx
15:     smem[2 · tid_inner] = $P[p_i]$ {Find the precomputed point}
16:     if $(a_i\&1)$ then
17:         smem[2 · tid_inner] = −smem[2 · tid_inner]
18:     end if
19:     smem[2 · tid_inner + 1] = PADD(smem[2 · tid_inner + 1], smem[2 · tid_inner])
20: end for
21: buffer[offset + num] = smem[2 · tid_inner + 1]
22: buffer_index[offset + num] = pre_bucket_idx
23: buffer_used[tid] = num + 1
```

Pippenger algorithm. Next, the bucket reduction within the M windows is completed. The MSM calculation is particularly bottlenecked by the bucket reduction stage of the Pippenger algorithm, which is essentially a sequential workload, as shown in Figure 3.1.6. To parallelize this stage, we propose Algorithm 6.

Algorithm 6 Cross-Window and Intra-Window Bucket Reduction

```
Require: Bucket vector $\vec{B}_{2^c-1} = [B_1, B_2, \ldots, B_{2^c-1}]$, window size $c$, compression coefficient $M$, thread count $n$, global memory $sos$
Ensure: MSM result $Q$
 1: Initialize $Q = 0$
 2: for each window group $g = 0$ to $\lceil \lambda/(Mc) \rceil - 1$ do
 3:     Initialize $Q_g = 0$
 4:     for each window $j = 0$ to $M - 1$ do
 5:         Initialize $Q_{g,j} = 0$
 6:         for each bucket $t = 1$ to $2^c - 1$ do
 7:             $Q_{g,j} = Q_{g,j} + t \cdot B_{g \cdot M + j, t}$
 8:         end for
 9:     end for
10:     $Q_g = \sum_{j=0}^{M-1} 2^{jc} \cdot Q_{g,j}$
11: end for
12: $Q = \sum_{g=0}^{\lceil \lambda/(Mc) \rceil - 1} 2^{gMc} \cdot Q_g$
13: Obtain thread processing boundaries $[s, e]$    // Each thread processes $(2^c-1+n-1)/n$ parts
14: $sos\_tmp = 0$; $st\_tmp = 0$; $tmp = 0$
15: for $i = e - 1$ to $s$ do
16:     $st\_tmp = PADD(st\_tmp, B[i])$
17:     $sos\_tmp = PADD(sos\_tmp, st\_tmp)$
18:     $tmp = mul(tid \cdot step\_len, st\_tmp)$
19:     $sos[tid] = PADD(sos\_tmp, tmp)$
20: end for
21: __syncthreads()
22: $Q = reduce\_sum(sos, n)$    // Parallel sum of $sos$
23: return $Q$
```

3.2 RcLboMMSM Design for Multi-GPU Systems

While previous research has primarily focused on optimizing single-device performance or introducing new protocols for distributed systems, these approaches often exhibit limitations in practical applications and face challenges in achieving universal scalability.

To overcome these limitations, we propose a hierarchical partitioning framework within our RcLboMMSM design, which extends to support multi-GPU systems. This framework divides the MSM task into multiple layers based on data scale and hardware characteristics. By employing an optimal sub-window search strategy, we efficiently allocate tasks within the computational resources of one or multiple GPUs, achieving better load balancing and performance. The RcLboMMSM system dynamically allocates tasks across GPUs based on data size and hardware resources, ensuring efficient utilization of all available computational resources. The hierarchical design and its extension to multi-GPU systems are illustrated in Figures 5

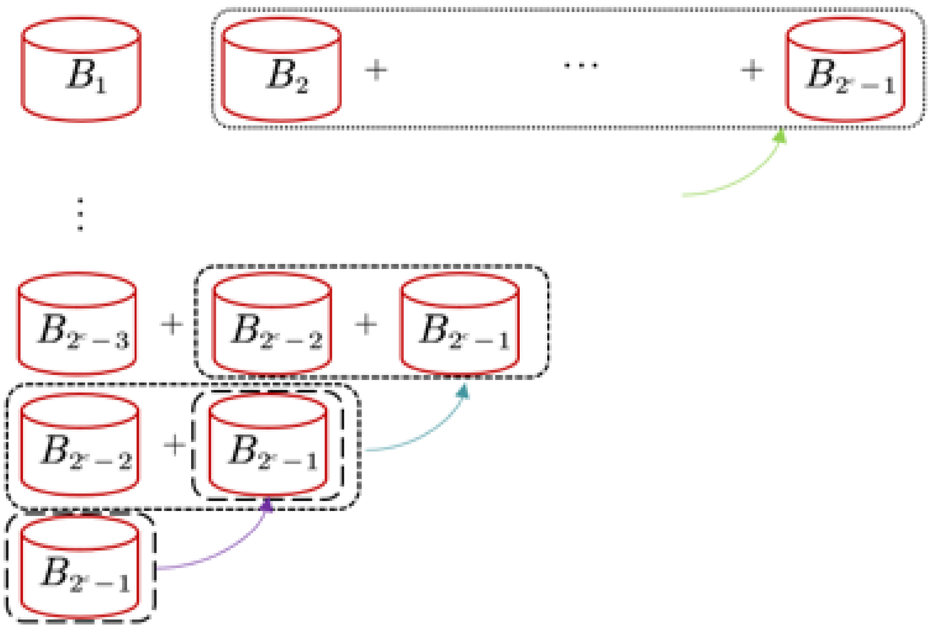


Figure 4: Serial Execution of Bucket Reduction

4 Evaluation

Our evaluation comprises three parts: assessing the overall performance of MSM operations on a single GPU, evaluating performance on multi-GPU systems, and providing a detailed analysis of the proposed optimizations. The experiments were conducted on three different GPU servers rented from the AutoDL platform, namely RTX 4090, V100, and A40. Table 3 presents the hardware specifications, including the CPU configuration. Since the MSM implementation primarily runs on the GPU, CPU performance mainly affects the scalar data transfer time between the host and device. To ensure a fair comparison, all benchmark implementations were executed on the same test platform. Although our framework supports load balancing across different graphics cards in a multi-GPU environment, due to limitations in our experimental setup, we conducted tests using multiple identical RTX 4090 GPUs.

Table 3: Hardware Configuration of Testing Environments

Environment	RTX 4090	V100	A40
SM Count	128	80	128
Memory	24GB	32GB	48GB
OS	Ubuntu 20.04	Ubuntu 20.04	Ubuntu 20.04
CUDA Version	11.8	11.8	11.8

We utilize the Zprize-provided testing framework, where the point set for evaluating multi-scalar multiplication (MSM) is randomly generated. There are two types of scalars: random scalars sampled from the scalar domain and clustered scalars as proposed by cuZK. In practical applications, scalars are not uniformly random; thus, evaluating MSM performance using only random scalars and point sets is insufficient. Therefore, we adopt a second evaluation standard using clustered scalars, limiting scalars to 32 distinct values

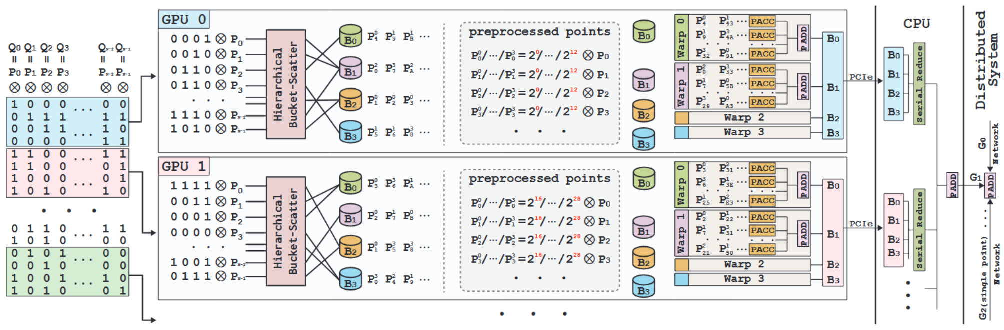


Figure 5: Hierarchical Partitioning Framework in RcLboMMSM Design

to reflect more realistic distributions.

We compare our implementation against several leading GPU-based MSM implementations (Table 4).

Table 4: Baseline GPU Implementations Used for Evaluation

Baseline	Supported Elliptic Curves	GPU Support
wlc-msm	BLS12-377, BLS12-381	Single GPU
Yrrid	BLS12-377	Single GPU
GZKP	ALT-BN128, BLS12-381, MNT4753	Single GPU
Bellperson	BLS12-381	Multi GPU
Sppark	BLS12-377, BLS12-381, BN254	Multi GPU
cuZK	BLS12-377, BLS12-381, MNT4753	Multi GPU

4.1 Overall Performance for sigle GPU

4.2 Overall Performance for sigle GPU

4.3 Breakdown Analysis

5 Conclusion