IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 522–544. DOI:10.46586/tches.v2024.i2.522-544

Load-Balanced Parallel Implementation on GPUs for Multi-Scalar Multiplication Algorithm

Yutian Chen, Cong Peng✉, Yu Dai, Min Luo✉ and Debiao He

School of Cyber Science and Engineering, Wuhan University, Wuhan, China.
wind.0xdktb@gmail.com,{cpeng,mluo}@whu.edu.cn

Abstract. Multi-scalar multiplication (MSM) is an important building block in most of elliptic-curve-based zero-knowledge proof systems, such as Groth16 and PLONK. Recently, Lu et al. proposed `cuZK`, a new parallel MSM algorithm on GPUs. In this paper, we revisit this scheme and present a new GPU-based implementation to further improve the performance of MSM algorithm. First, we propose a novel method for mapping scalars into Pippenger's bucket indices, largely reducing the number of buckets compared to the original Pippenger algorithm. Second, in the case that memory is sufficient, we develop a new efficient algorithm based on homogeneous coordinates in the bucket accumulation phase. Moreover, our accumulation phase is load-balanced, which means the parallel speedup ratio is almost linear growth as the number of device threads increases. Finally, we also propose a parallel layered reduction algorithm for the bucket aggregation phase, whose time complexity remains at the logarithmic level of the number of buckets. The implementation results over the BLS12-381 curve on the `V100` graphics card show that our proposed algorithm achieves up to 1.998×, 1.821× and 1.818× speedup compared to `cuZK` at scales of 2^{21}, 2^{22}, and 2^{23}, respectively.

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

1 Introduction

In recent years, there has been a growing emphasis on privacy concerns within the industrial sector. Zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK), an excellent cryptographic primitive, not only provides robust privacy protection but also allows for essential audits. It has been widely used in industrial-grade solutions such as anonymous transactions in Zerocash [BSCG+14] and flexible anonymous credentials zk-cred [RWGM23]. It allows the prover to generate a proof π for any (small) non-deterministic polynomial (NP) relation $R = \{(x; w) : P(x, w)\}$. Unfortunately, most of elliptic-curve-based zk-SNARKs like Groth16 [Gro16] and PLONK [GWC19] still suffer from performance bottlenecks. Various acceleration solutions have already been published. For instance, Ni et al. [NZ23] took advantage of GPU to enhance the efficiency of zkSNARK by accelerating the Number-Theoretic Transform (NTT) and Inverse Number-Theoretic Transform (INTT). We notice that the computational cost of the *Setup* or *Prove* phase is significantly influenced by the number of circuit multiplication gates, resulting in a much longer time duration compared to the *Verify* phase. Furthermore, large-scale Multi-scalar multiplication (MSM) operations occupy the majority of the computational cost during the proof generation phase. Thus, the practical deployment of zk-SNARKs urgently requires fast MSM computational algorithms.

In scenarios involving zk-SNARKs or large-scale BLS signature aggregation [BGLS03], the state-of-the-art serial algorithms for MSM are the Pippenger algorithm and its vari-

Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

ants [Pip76, BDLO12]. These algorithms partition the λ-bit scalar into multiple c-bit windows. Subsequently, all points are sorted out into buckets with respect to sub-scalar values, and finally the values within buckets are aggregated to derive the final result. Recently, a series of works were proposed to further improve the performance of the Pippenger algorithm and its variants. Botrel and Housni [BEH23] optimized the arithmetic of finite fields by improving on the Coarsely Integrated Operand Scanning (CIOS) modular multiplication and proposed a new coordinate system for twisted Edwards curves tailored for the Pippenger algorithm. Luo, Fu and Gong [LFG23] proposed a bucket set construction to speed up MSM over fixed points with the help of large precomputation tables.

Owing to the thriving development of modern GPU architectures, researchers have proposed several GPU-accelerated implementations based on the Pippenger algorithm. `MatterLab` and `Yrrid` [Mat22, Yrr22] are the winners of the Zprize competition [Zpr22], a competition which focuses on accelerating MSM using GPUs. Both `MatterLab` and `Yrrid` utilized radix sort to process the scalars used in MSM. Recently, Lu et al. [LWY+23] proposed `cuZK`: a new GPU-accelerated implementation. In particular, the authors pointed out the possibility of load imbalance for the implementations in `Matterlab` and `Yrrid`. For this reason, they also converted the major operations used in the Pippenger algorithm to a series of basic sparse matrix operations, including sparse matrix transpose and sparse matrix-vector multiplication. In fact, `cuZK` is well-suit for the high parallelism in GPU-based implementation and has nearly perfect linear speedup over the Pippenger algorithm.

1.1 Our contributions

In this paper, we propose a high-speed GPU-based implementation for MSM in scenarios with different memory sizes. The proposed implementations are applicable to most of curve-based zk-SNARKs and achieve high performance on modern GPU architectures. Our contributions are summarized as follows:

- In Section 3.1, we propose a new method for mapping scalars into Pippenger's bucket indices, reducing the number of buckets to $\frac{1}{4}$ of that in the original Pippenger algorithm. To be specific, we fix the Pippenger's window size at c-bit and convert each $k_{i,j} \in [0, 2^c]$ into 0 or a unique odd bucket index $\bar{k}_{i,j} \in [1, 2^c - 1]$. We recode the bucket indices along with the mapped additional information for point operations during the bucket accumulation phase. This custom encoding format does not increase the sorting cost when we use the radix sort algorithm [Pow90].

- In Section 3.2, we focus on the fundamental operations of point addition during the bucket accumulation phase. We employ the mixed point addition formulas in cached Jacobian coordinates [CC86] when the GPU memory is of typical size and switch to a more efficient algorithm based on homogeneous coordinates when the memory is sufficient. Compared to the former, for every two points added to the same bucket, the accumulation algorithm based on homogeneous coordinates saves one finite field multiplication cost (-5%). In addition, we also apply the technique of lazy reduction [AKL+11, Sco07] to all point addition and doubling operations and employ certain CUDA assembly tricks to further enhance speed, etc.

- In Section 3.3, we present our load-balanced bucket accumulation method, which means the parallel speedup ratio is almost linear growth as the number of device threads increases. Differing from `cuZK`, our approach abstains from the use of data structures like vectors (libstl-cuda/vector), which could potentially involve dynamic memory allocation. Instead, we design a compact static buffer and employ binary sieving to accumulate results from different threads.

- In Section 3.4, we provide a parallel layered reduction algorithm for the serially executed bucket aggregation phase. The time complexity of our algorithm remains at the logarithmic level of the number of buckets.

- In Section 3.5, we discuss the computational cost for each phase within each thread. In addition, we analyze why our implementation is not constant-time, and illustrate a new algorithm for bucket accumulation to remedy this shortcoming in Section 5.

- In Section 4, we evaluate our implementation on two SNARK-friendly curves, BLS12-381 and BLS24-315 [EHG22].

  - In particular, the evaluation results show that our MSM achieves 1.998×, 1.821×, and 1.818× speedup on `cuZK` at scales of 2^{21}, 2^{22}, and 2^{23}, respectively (on NVIDIA V100-SXM2 GPU card with BLS12-381). On higher-powered GPU cards like RTX 4090, our speedup ratio is also nearly linear growth as the number of device threads increases (1.81×, 1.49×, and 1.42× speedup on `cuZK` at scales of 2^{21}, 2^{22}, and 2^{23}, respectively).
  - On the BLS24-315 curve, our implementation based on homogeneous coordinates achieves 90.35ms (on RTX 4090 with 2^{23} points MSM). Furthermore, since the bucket accumulation algorithm in previous work assigns non-overlapping bucket indices to each thread, we implement this previous method to demonstrate the superiority of our proposed load-balanced algorithm. The comparison reveals up to 1.853× speedup.

2 Preliminaries

Notation. Let $[n]$ denote the set of integers $\{1, 2, \cdots, n\}$ for $n \in \mathbb{N}$. For a λ-bit integer σ, we write that $\sigma = \sum_{i=0}^{\lambda-1} \sigma[i] \cdot 2^i$, where $\sigma[i]$ is the i-th bit of σ. And, we write $\sigma[s:e] = \sum_{i=s}^{e} \sigma[i] \cdot 2^{i-s}$ where $0 \leq s \leq e < \lambda$.

2.1 Elliptic point groups and coordinates

Let $\mathbb{F}_p$ denote the finite field of the prime order p. $|\mathbb{F}_p|$ denotes the byte length of elements in $\mathbb{F}_p$. The group $\mathbb{G}$ is a subgroup of the elliptic points group $E(\mathbb{F}_p)$ over $\mathbb{F}_p$ with the order r. In the group $\mathbb{G}$, the identity element $\mathcal{O}$ refers to the point at infinity. The group law **PADD** and **PDBL** refer to the point addition for unequal points and point doubling for equal points, respectively. $|\mathbb{G}|$ denotes the byte length of elements in $\mathbb{G}$. Let $\lambda = \lceil \log_2 r \rceil$ denote the bits of the order r.

Different coordinates. To reduce storage overhead, elliptic curve points are commonly represented in affine coordinates, i.e., only two field elements $\{X, Y\}$ are required to represent one point. To speed up computations, diverse coordinate systems and addition formulas have emerged [1], such as Jacobian coordinates $\{X, Y, Z\}$ $(x = X/Z^2, y = Y/Z^3)$, modified Jacobian coordinates $\{X, Y, Z, T\}$ $(x = X/Z^2, y = Y/Z^3, T = aZ^4)$, homogeneous coordinates $\{X, Y, Z\}$ $(x = X/Z, y = Y/Z)$, and cached Jacobian coordinates $\{X, Y, ZZ, ZZZ\}$ $(x = X/ZZ, y = Y/ZZZ, ZZ^3 = ZZZ^2)$. For subsequent optimisation work, we review computational overhead under several coordinate systems in Table 1. Traditionally, the single-scalar multiplication approach is said to use mixed point addition between affine and Jacobian coordinates and point double under Jacobian coordinates.

[1] Detailed information can be found in https://hyperelliptic.org/EFD/index.html

Table 1: Costs of point addition in different coordinate systems. The notations **M** and **S** represent the costs of multiplication and squaring in $\mathbb{F}_p$, respectively ($\mathbf{M} \approx \mathbf{S}$).

Coordinate systems	Mixed addition (one in affine)	Addition (both in affine)	Addition (both in projective)
Jacobian	$7\mathbf{M} + 4\mathbf{S}$	$4\mathbf{M} + 2\mathbf{S}$	$11\mathbf{M} + 5\mathbf{S}$
Modified Jacobian	$7\mathbf{M} + 6\mathbf{S}$	$3\mathbf{M} + 4\mathbf{S}$	$11\mathbf{M} + 7\mathbf{S}$
Cached Jacobian	$8\mathbf{M} + 2\mathbf{S}$	$4\mathbf{M} + 2\mathbf{S}$	$12\mathbf{M} + 2\mathbf{S}$
Homogeneous	$9\mathbf{M} + 2\mathbf{S}$	$5\mathbf{M} + 2\mathbf{S}$	$12\mathbf{M}$

2.2 Multi-Scalar Multiplication

Given n scalars k_i and n points $P_i \in \mathbb{G}$ for $i \in [n]$, Multi-Scalar Multiplication (MSM) computation process is to compute $Q = \sum_{i=1}^{n} k_i P_i$, where $0 \leq k_i < r$. As shown in **Alg**. 1, the double-and-add method can be used as a straightforward approach to compute MSM, which needs to perform $n\lambda$ **PADD** and $\lambda - 1$ **PDBL** in the worst case. This would be prohibitively slow and thus unacceptable for real-world applications in zk-SNARKs.

Algorithm 1 The double-and-add algorithm for computing MSM

Input: The scalars $\{k_i\}_{i\in[n]}$ and the points $\{P_i\}_{i\in[n]} \in \mathbb{G}$
Output: The output point $Q = \sum_{i=1}^{n} k_i P_i$

```
Q = 𝒪
for j = λ − 1 to 0 by 1 do
    Q = PDBL(Q)
    for i = 1 to n by 1 do
        if k_i[j] = 1 then
            Q = PADD(Q, P_i)
        end if
    end for
end for
return Q
```

2.3 The Pippenger Algorithm

For the large-scale MSM, the Pippenger algorithm and its variants [Pip76, BDLO12] are the most popular serial algorithms. Our proposed GPU-accelerated MSM algorithm is also built on this foundation algorithm. The Pippenger algorithm is described as follows:

Step-1: Decompose the main task into multiple subtasks. The Pippenger algorithm first chooses an integer $c \in [\lambda]$ as the window size, and decomposes each λ-bit scalar k_i into multiple c-bit scalar slices $k_{i,j} = k_i[j \cdot c : j \cdot c + c - 1]$ such that $k_i = \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc}$. Thus, the main task can be considered as computing $\lceil \frac{\lambda}{c} \rceil$ subtasks, called the smaller-scale MSM, i.e., $Q_j = \sum_{i=1}^{n} k_{i,j} P_i$ where $0 \leq k_{i,j} < 2^c$ and $\lambda_c = \lceil \frac{\lambda}{c} \rceil - 1$.

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{j=0}^{\lambda_c} k_{i,j} 2^{jc} P_i = \sum_{j=0}^{\lambda_c} 2^{jc} Q_j.$$

Step-2: Compute the bucket points in each subtask. To complete each subtask, the Pippenger algorithm introduces a buffer point called *"bucket"* and divides the subtask into two phases. In the first phase for each subtask Q_j, called *bucket accumulation*, it

computes $2^c - 1$ "*bucket*" points $\{B_t\}_{1 \le t \le 2^c-1}$, which is the sum of all points with the small scalar equaled to t, i.e., $B_t = \mathbf{SUM}\{P_i | k_{i,j} = t\}$ as shown in Figure 1. Note that $B_0 = \mathcal{O}$ since it is the sum of all points with zero scalars. In the second phase for each subtask Q_j, called *bucket aggregation*, the Pippenger algorithm sequentially computes $G_{t-1} = G_t + B_{t-1}$ for $G_{2^c-1} = B_{2^c-1}$ to G_1 as shown in Figure 2. So, we have

$$Q_j = \sum_{i=1}^{n} k_{i,j} P_i = \sum_{t=1}^{2^c-1} tB_t = \sum_{t=1}^{2^c-1} G_t. \tag{1}$$

After calculating all points $\{G_t\}_{1 \le t \le 2^c-1}$, it can accumulate them to obtain Q_j.

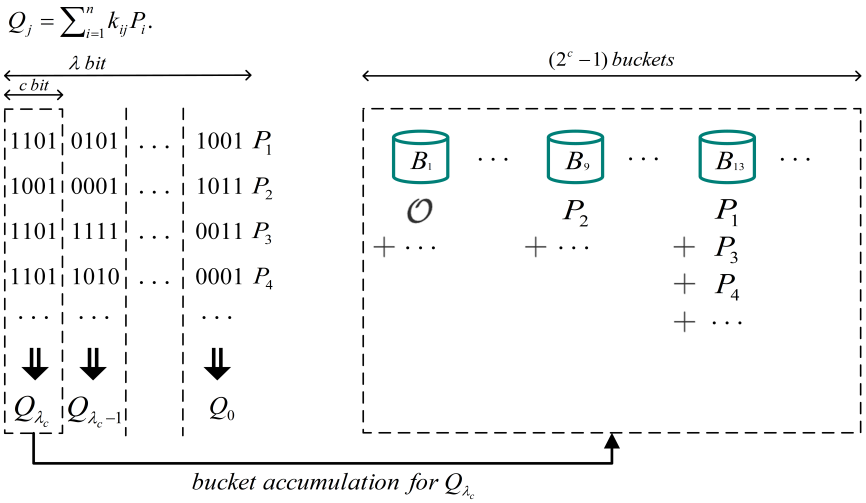

Figure 1: An example of the bucket accumulation phase.

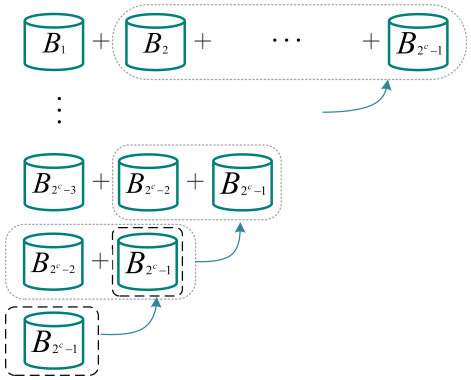

Figure 2: The original bucket aggregation phase (excluding the final summation).

Step-3: Aggregate all subtask results into the main task result. Finally, the Pippenger algorithm sets $Q = Q_{\lambda_c}$ as the initial status and computes $Q = 2^c Q + Q_i$ from $i = \lambda_c$ to 0. After $\lceil \frac{\lambda}{c} \rceil$ rounds, it can get the final result Q.

Computational cost. It can be observed that the Pippenger algorithm converts all scalar multiplications into **PADD** and **PDBL** computations. The computational costs of scalar segmentation and sorting in the preceding steps are relatively small, with the primary time consumption occurring in the subsequent point operations. For each subtask, the bucket

accumulation phase requires at most n **PADDs** to add base points to the buckets, and the bucket aggregation phase requires $(2^{c+1}-4)$ **PADDs** to add buckets to subtask sum. Finally, the subtask aggregation requires $c(\lceil\frac{\lambda}{c}\rceil-1)$ **PDBLs** and $\lceil\frac{\lambda}{c}\rceil$ **PADDs**. Thus, the Pippenger algorithm needs to perform $\lceil\frac{\lambda}{c}\rceil(n+2^{c+1}-3)$ **PADDs** and $c(\lceil\frac{\lambda}{c}\rceil-1)$ **PDBLs**.

2.4 Challenges in parallel implementations

CUDA programming model. Modern GPU architecture is characterized by its highly parallel and massively multithreaded design, making it well-suited for computationally intensive tasks. It consists of thousands of small processing cores organized into Streaming Multiprocessors (SMs). These cores are designed for Single Instruction, Multiple Thread (SIMT) execution, which means they can execute multiple threads simultaneously. GPU memory is divided into global memory, shared memory, and local memory, each with its own characteristics and access patterns. Efficient memory management is crucial for optimizing GPU performance.

The CUDA programming model is a parallel computing platform and API developed by NVIDIA for GPUs. It allows developers to harness the computational power of GPUs for a wide range of tasks. In this model, programmers write kernels, which are parallel functions that can be executed by thousands of GPU threads. These kernels are launched from the CPU and executed on the GPU.

For bucket accumulation. As shown in Figure 1, each subtask requires a buffer size of 2^c-1 elliptic curve points, which is entirely acceptable in any mode. However, for GPU-accelerated mode, there still exists a critical influencing factor in the accumulation phase. Assuming the total number of threads that the GPU card can provide is N, the fundamental idea of the parallel implementation is that each thread processes a consecutive set of $\lceil\frac{n}{N}\rceil$ sub-scalars along with their associated points. But when different threads process buckets stored at the same storage in the buffer, write conflicts are inevitably encountered. Previous works like [6bl22, Mat22] either employs performance-degrading atomic functions or utilizes Alg. 2. Before Alg. 2, they initialize an array of tuple pairs consisting of sub-scalars and point indices $\{(k_{i,j},i)\}$ and perform radix sorting based on the keys $k_{i,j}$ to obtain a new sorted array of tuple pairs $\{(a_i,p_i)\}$ (in ascending order). Then, each thread no longer simply processes data with indices $\in[s,e)=[\lceil\frac{n}{N}\rceil\cdot tid,\lceil\frac{n}{N}\rceil\cdot(tid+1))$, but instead adjusts the left edge to $min\{s'\mid s'\geq s\wedge a_{s'}\neq a_{s'-1}\}$ (excluding the case where $s=0$), and similarly adjusts the right edge (exclusive) to $min\{e'\mid e'\geq e\wedge a_{e'}\neq a_{e'-1}\}$. This ensures that write operations of buckets between threads do not conflict. However, this approach brings about the issue of load imbalance and may even result in threads idling, leading to a certain degree of computing power waste.

For bucket aggregation. The bucket aggregation phase can be divided into two stages: the bucket updates and the final summation. The final summation can be accomplished using the well-known parallel reduction algorithms. However, it is evident that the bucket updating part (in Figure 2) is a serial computational process, for which no practical parallel algorithms have been proposed at present (In prior works, the parallel methods used for this part always involved small-scale scalar multiplication, which resulted in a performance impact).

3 Our GPU-Accelerated MSM Algorithm

In this section, we describe our specific implementation of an individual Pippenger subtask on GPU platforms. We use N to denote the maximum thread number supported by the current GPU device.

Algorithm 2 Previous parallel bucket accumualtion algorithm

Input: Sorted array (ascending) of tuple pairs $\{(a_i, p_i)\}_{i\in[n]}$, points $\{P_i\}_{i\in[n]}$
Output: Buckets $\{B_t\}$ $(0 \le t < 2^c)$, which is initialized as $\{\mathcal{O}\}$ beforehand

1: $s = \lceil \frac{n}{N} \rceil \cdot tid$
2: $e = \lceil \frac{n}{N} \rceil \cdot (tid + 1)$ ▷ $tid \in [0, N)$ is the index of thread
3: **if** $s \ge n$ **then** return
4: **if** $e \ge n$ **then** $e = n$
5: **while** $s \ne 0$ **and** $s < n$ **and** $a_s = a_{s-1}$ **do**
6: $\quad s = s + 1$
7: **end while**
8: **while** $e < n$ **and** $a_e = a_{e-1}$ **do**
9: $\quad e = e + 1$
10: **end while**
11: **for** $i = s$ **to** $e - 1$ **by** 1 **do**
12: $\quad B_{a_i} =$**PADD**(B_{a_i}, P_{p_i})
13: **end for**

3.1 Scalar processing and point precomputation

The first optimization is to reduce the number of buckets from 2^c-1 to $2^{c-2}-1$ by processing all scalars slices $\{k_{i,j}\}_{1\le i\le n, 0\le j\le \lambda_c}$ for each subtask. This could make the number of buckets a quarter of the number in the original Pippenger algorithm. Mathematically, the scalar slices are transformed according to the following sequence:

$$k_{i,j}P_i \to \tilde{k}_{i,j} \cdot (-1)^{s_{i,j}} P_i \to \bar{k}_{i,j} \cdot (-1)^{s_{i,j}} 2^{h_{i,j}} P_i \to \bar{k}_{i,j} \cdot (-1)^{s_{i,j}} P'_{i,h_{i,j}}$$

where $\bar{k}_{i,j}$ is an odd number of at most $c-1$ bits and $P'_{i,h_{i,j}}$ is a precomputed point.

Algorithm 3 Scalar conversion to the float representation

Input: The bit-length λ, n integers $\{k_i\}_{i\in[n]}$ and the window size $c \in [\lambda]$
Output: The integer tuples $\{\bar{k}_{i,j}, h_{i,j}, s_{i,j}\}$

1: $s_{i,-1} = 0$, $t[0] = 0$, $t[2] = 2^c$
2: **for** $j = 0$ **to** λ_c **by** 1 **do**
3: $\quad k_{i,j} = k_i[jc : jc + c - 1]$
4: $\quad t[1] = k_{i,j} + s_{i,j-1}$ ▷ $k'_{i,j} = t[1]$
5: $\quad s_{i,j} = (t[1] >> c) | (t[1] >> (c-1))$
6: $\quad \tilde{k}_{i,j} = t[s_{i,j} + 1] - t[s_{i,j}]$ ▷ $\tilde{k}_{i,j} = (s_{i,j})\ ?\ (2^c - k'_{i,j}) : k'_{i,j}$
7: $\quad h_{i,j} = max\{\eta : 2^\eta \mid \tilde{k}_{i,j}\}$, $\bar{k}_{i,j} = \tilde{k}_{i,j} >> \eta$ ▷ factor $\tilde{k}_{i,j}$
8: **end for**

- **Convert scalars to the signed representation.** Commonly, we can convert $k_i \in [0, r)$ from its unsigned 2^c-ary representation into the signed representation with each digit in the range of $[-2^{c-1}, 2^{c-1}]$, that is $k_i = \sum_{j=0}^{\lambda_c} k_{i,j} = \sum_{j=0}^{\lambda_c} \tilde{k}_{i,j} \cdot (-1)^{s_{i,j}}$. Let $s_{i,-1} = 0$. From $j = 0$ to λ_c, one can compute $k'_{i,j} = k_{i,j} + s_{i,j-1}$ and set $s_{i,j} = 1$, $\tilde{k}_{i,j} = 2^c - k'_{i,j}$ if $k'_{i,j} \ge 2^{c-1}$ or $s_{i,j} = 0$, $\tilde{k}_{i,j} = k'_{i,j}$ otherwise. As long as the curve parameters are appropriate, here $s_{i,\lambda_c} = 0$ [2]. Then, the sign bit $s_{i,j}$ can be stored for each scalar $k_{i,j}$. Note that if $k'_{i,j} = 2^{c-1}$, we have $s_{i,j} = 1$ and $\tilde{k}_{i,j} = 2^{c-1}$.

[2] If c is a factor of λ, there is an extreme case such that $\tilde{k}_{i,\lambda_c} = 2^{c-1}$. This causes the number of buckets in the λ_c-th subtask to be increased by 1. But, for most SNARK-friendly curves, including BLS12-377, BLS12-381, and BLS24-315 [EHG22], it is not going to happen.

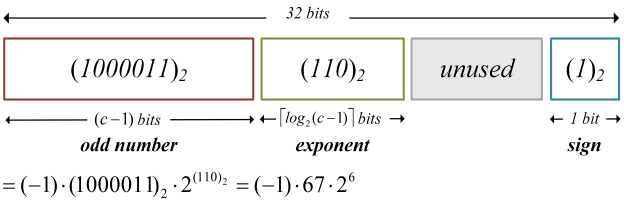

Figure 3: Example of the 32-bit integer $\hat{k}_{i,j}$.

- **Convert scalars to the float representation.** Inspired by the floating-point format, we can compute an odd number $\bar{k}_{i,j} \in [0, 2^{c-1})$ and a small exponent $h_{i,j} \in [0, c-1]$ such that $\tilde{k}_{i,j} = \bar{k}_{i,j} 2^{h_{i,j}}$. Then, we can reorganize an integer $\hat{k}_{i,j}$ with the higher $c-1$ bits to store $\bar{k}_{i,j}$, the middle $\lceil \log_2(c-1) \rceil$ bits to store $h_{i,j}$ and the least significant bit to store $s_{i,j}$. We provide an example of the format of $\hat{k}_{i,j}$ in Figure 3. At maximum, a 32-bit integer can store $\hat{k}_{i,j}$ for any window size $c \leq 26$. Note that we set $\hat{k}_{i,j} = 0$ if $\tilde{k}_{i,j} = 0$. For simplicity, we write $\hat{k}_{i,j}.ODD$, $\hat{k}_{i,j}.EXP$ and $\hat{k}_{i,j}.SIGN$ to denote the odd number, exponent and sign bit of $\hat{k}_{i,j}$, respectively.
- **Create a scalar mapping table.** Typically, we can compute the integer tuples $\{\bar{k}_{i,j}, h_{i,j}, s_{i,j}\}$ by Alg. 3 and reorganize them into $\hat{k}_{i,j}$. If c is not particularly large (e.g. no more than 16), we can create a table with the input $k_{i,j}$ and output $\hat{k}_{i,j}$. This table costs 2^{c+2} bytes of storage but can improve scalar processing performance.

Change the way to add points into the bucket. After scalar processing, we should add the point $(-1)^{s_{i,j}} P'_{i,h_{i,j}} = (-1)^{s_{i,j}} 2^{h_{i,j}} P_i$ to the bucket $B_{\bar{k}_{i,j}}$ with the index $\bar{k}_{i,j}$. The point $-P'_{i,h_{i,j}}$ is easy to compute by the negation of y-coordinate. But, calculating from P_i to the point $2^{h_{i,j}} P_i$ requires $h_{i,j}$ times **PDBLs**. For efficiency, points $\{2^1 P_i, 2^2 P_i, \cdots, 2^\tau P_i\}$ are precomputed with a threshold parameter $\tau \in [1, c-1]$ and stored in the GPU memory with $\tau \cdot n$ points. Obviously, different $h_{i,j}$ can be greater or less than the threshold τ. So,

- If $h_{i,j} \leq \tau$, $P'_{i,h_{i,j}}$ can be looked up directly in precomputed point table.
- If $h_{i,j} > \tau$, $2^\tau P_i$ can be looked up directly in precomputed point table. Then, it is also necessary to recalculate $P'_{i,h_{i,j}} = 2^{h_{i,j}-\tau} \cdot 2^\tau P_i$ with $h_{i,j} - \tau$ times **PDBLs**. Statistically, the values of $\tilde{k}_{i,j}$ that satisfy $\{2^\tau \nmid \tilde{k}_{i,j} \wedge 2^{\tau+1} \mid \tilde{k}_{i,j}\}$ constitute only a fraction of approximately $1/2^{\tau+1}$ of the total, which is absolutely acceptable in the parallel program.

Totally, in this optimization phase, we need 2^{c+2} bytes memory for scalar processing, $\tau \cdot n|\mathbb{G}|$ bytes memory for precomputed points with $\tau \cdot n$ times **PDBL** operations. When τ equals to the maximum value $c-1$, this process will consist of only one table lookup and one negative operation in $\mathbb{F}_p$. Thus, it is constant-time for any tuple $(k_{i,j}, P_i)$ processing.

3.2 Single-point and double-point addition optimization

The second optimization is to reduce the number of multiplications in the process of adding one or two original points into one bucket. The basic idea is still to represent buckets in a specific coordinate system, and then accumulate the points in affine coordinates into the bucket. We mainly design two approaches:

- **Cached Jacobian coordinates.** Similar to prior works [Mat22, LWY+23], we can add a single point to buckets in cached Jacobian coordinates, which only needs 10 **M** per one point. As shown in Table 1, this coordinate system is the optimum.

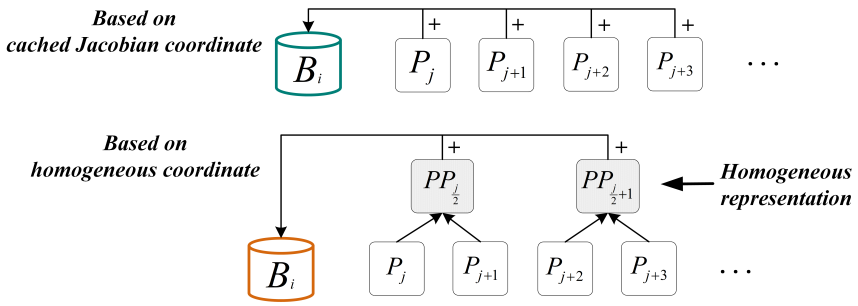

Figure 4: Two methods of adding affine points to the bucket.

- **Homogeneous coordinates.** Fortuitously, we find that adding two points to the same bucket per round further reduces the number of multiplications. For instance, we choose homogeneous coordinates in Table 1, add two affine points (P_j, P_{j+1}) to a buffer point $PP_{j/2}$ (cost 5 $\mathbf{M}$ + 2 $\mathbf{S}$) [CMO98], then add $PP_{j/2}$ with the bucket via point addition formula in homogeneous coordinates (cost 12 $\mathbf{M}$).

Remark that this double-point point approach only costs $17\mathbf{M} + 2\mathbf{S}$ ($\approx 19\mathbf{M}$), making it approximately 5% faster than the single-point approach ($16\mathbf{M} + 4\mathbf{S} \approx 20\mathbf{M}$) per two points. However, the double-point point approach is non-constant time since the number of multiplications per point depends on the parity of the number of points added to the same bucket.

We also employ the lazy reduction technique [AKL+11, Sco07] in our point addition and point doubling operations. More specifically, we reduce the number of Montgomery reductions [Mon85, KKAK96] required for point addition (both points are in projective), mixed addition (one point is in affine), and doubling in the cached Jacobian coordinate system by one. Similarly, for point addition (both points are in projective), addition (both points are in affine), and doubling in the homogeneous coordinate system, we reduce the number of Montgomery reductions by 3, 1, and 1, respectively.

3.3 Load-balanced accumulation from original points to buckets

The third optimization is to accumulate all original points into buckets using parallel threads. Let *tid* denote each thread's index. For the maximum thread number N, we assign each thread to process $\lceil \frac{n}{N} \rceil$ points with indices $\in [s, e) = [\lceil \frac{n}{N} \rceil \cdot tid, \lceil \frac{n}{N} \rceil \cdot (tid + 1))$. However, *unordered and random scalars can lead to a conflict where different threads write to the same bucket.* Alg. 2 aims to adjust the left and right boundaries (s, e) such that buckets processed by each thread do not overlap. However, such a solution may lead to an imbalanced thread workload, resulting in a waste of computational resources. Therefore, we pre-allocating non-conflicting point buffers to solve this problem.

Instead of writing to buckets directly, we write to non-conflicting static buffers assigned to each thread. Intuitively, each thread handles data corresponding to at most $\lceil \frac{n}{N} \rceil$ buckets. We can allocate a buffer of n points for each thread. However, this approach would incur a memory penalty. Moreover, if we further parallelize with $\lceil \frac{\lambda}{c} \rceil$ subtasks, a memory size of $\lceil \frac{\lambda}{c} \rceil \cdot n$ points, which is sparse, becomes unacceptable. Therefore, we preallocate a static buffer for each of the $\lceil \frac{\lambda}{c} \rceil$ subtasks, but we set the starting offset within the respective buffer where each thread writes to as:

$$offset_{tid} = tid + min \left\{ \left\lfloor \frac{\hat{a}_i.ODD + 1}{2} \right\rfloor \right\}_{i \in [s,e)}, \tag{2}$$

where $\lfloor \frac{\hat{a}_i.ODD+1}{2} \rfloor$ is the bucket offset corresponding to $\hat{a}_i$. The correctness of $offset_{tid}$ is evident because the previous thread indices $\in [0, tid)$ correspond to at least tid points. Additionally, whenever the boundary between different bucket offsets falls within a thread rather than between different threads, it results in an extra point. For clarity, we denote such boundaries within threads as "inner-boundary" and between threads as "inter-boundary", as shown in Figure 5. Since $\{\hat{a}_i\}$ are in ascending order, the minimum bucket offset from the current threads is exactly $\lfloor \frac{\hat{a}_s.ODD+1}{2} \rfloor$, which is also the maximum number of inner-boundaries existing in previous threads within the same Pippenger subtask. It ensures that the starting offset never exceeds $offset_{tid}$.

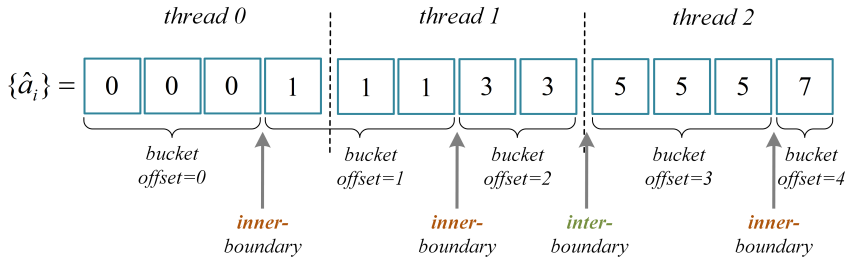

Figure 5: Example of "inner-boundary" and "inter-boundary" between different bucket offsets.

In summary, we deduce that the total size of the secure static buffer is $\lceil \frac{\lambda}{c} \rceil \cdot (N + 2^{c-2})$, and the points within it are compact (with unused memory occurring only as a low probability event when the boundaries happen to fall between threads). In large-scale MSM computations, such a buffer size is significantly smaller than $\lceil \frac{\lambda}{c} \rceil \cdot n$ and does not increase with the growth of n. It is solely dependent on the characteristics of the device.

In addition to allocating the buffer containing elliptic curve points, we also need to allocate three auxiliary arrays: *buffer_offset*, *buffer_index*, and *buffer_used*. These arrays are respectively used to store the starting positions where each thread writes to the buffer (as shown in Eq. (2)), the bucket offsets corresponding to the points being written, and the actual buffer size utilized by each thread. Clearly, the sizes of these three auxiliary arrays are $\lceil \frac{\lambda}{c} \rceil \cdot N$, $\lceil \frac{\lambda}{c} \rceil \cdot (N + 2^{c-2})$, and $\lceil \frac{\lambda}{c} \rceil \cdot N$, respectively. They store simple integer elements that are much smaller in size compared to the curve points.

Then we can use a three-step process to complete the bucket accumulation phase:

1) Sorting the processed scalars. For each subtask, we initialize an array consisting of converted scalars and point indices in the form of tuples $\{(\hat{k}_{i,j}, i)\}$ before bucket accumulation, and sort them in ascending order based on the key $\hat{k}_{i,j}.ODD$. For simplicity, we collectively refer to the ordered arrays obtained from j subtasks as $\{(\hat{a}_i, p_i)\}$.

2) Accumulating parts of the buckets into buffers. With the non-conflicting static buffers, we accumulate parts of the buckets into *buffer* in each thread and record the corresponding values in three auxiliary arrays *buffer_offset*, *buffer_index*, and *buffer_used*. Furthermore, we utilize the shared memory, denoted as $smem[2 \cdot NTHREADS]$ (or $smem[3 \cdot NTHREADS]$), to speed up read and write operations on original/bucket points. Here, $NTHREADS$ is the (CUDA) block size. As shown in Alg. 4, each thread writes $smem[2 \cdot tid_{inner} + 1]$ back to global memory after processing each bucket in the cached Jacobian coordinates, while $smem[3 \cdot tid + 2]$ is written back to the global memory in homogeneous coordinate systems.

3) Aggregating buffered points into buckets. After writing to the buffers, we need to aggregate the buffered points back into their respective unique buckets. It's worth noting that the bucket offsets corresponding to the points stored in the buffer are also sorted in ascending order. Therefore, we can utilize a variation of the binary search [Wil76]

Algorithm 4 Accumulation of bucket parts into buffers using the shared memory

Input: Sorted array (ascending) of tuple pairs $\{(\hat{a}_i, p_i)\}_{i\in[n]}$, points $\{P_i\}_{i\in[n]}$, thread index tid, intra-block thread index tid_{inner}
Output: $buffer$,$buffer\_offset$,$buffer\_index$,$buffer\_used$

Obtain the boundaries (s, e)
$pre_bucket_idx = 0$x8000 ▷ 0x8000: non-existent bucket index
$buffer\_offset[tid] = offset = tid + \lfloor \frac{\hat{a}_s.ODD+1}{2} \rfloor$
$num = 0$
◇ **Cached Jacobian-based:**
$smem[2 \cdot tid_{inner} + 1] = \mathcal{O}$
for $i = s$ **to** $e - 1$ **by** 1 **do**
  if $\hat{a}_i.ODD \neq pre\_bucket\_idx \wedge i \neq s$ **then**
    $buffer[offset + num] = smem[2 \cdot tid_{inner} + 1]$
    $buffer\_index[offset + num] = \lfloor \frac{pre\_bucket\_idx+1}{2} \rfloor$
    $smem[2 \cdot tid_{inner} + 1] = \mathcal{O}$, $num = num + 1$
  end if
  $pre\_bucket\_idx = \hat{a}_i.ODD$
  $smem[2 \cdot tid] = 2^{MIN(\tau, \hat{a}_i.EXP)} P_{p_i}$ ▷ precomputed
  for $j = 1$ **to** $\hat{a}_i.EXP - \tau$ **by** 1 **do**
    $smem[2 \cdot tid_{inner}] =$**PDBL**$(smem[2 \cdot tid_{inner}])$
  end for
  if $\hat{a}_i.SIGN = 1$ **then**
    $smem[2 \cdot tid_{inner}] = -smem[2 \cdot tid_{inner}]$
  end if
  $smem[2 \cdot tid_{inner} + 1] =$**PADD**$(smem[2 \cdot tid_{inner} + 1], smem[2 \cdot tid_{inner}])$
end for
$buffer[offset + num] = smem[2 \cdot tid_{inner} + 1])$
$buffer\_index[offset + num] = \lfloor \frac{pre\_bucket\_idx+1}{2} \rfloor$
$buffer\_used[tid] = num + 1$
◇ **Homogeneous-based:**
$smem[3 \cdot tid_{inner} + 2] = \mathcal{O}$, $parity = 0$
for $i = s$ **to** $e - 1$ **by** 1 **do**
  if $\hat{a}_i.ODD \neq pre\_bucket\_idx \wedge i \neq s$ **then**
    if $parity \neq 0$ **then**
      $smem[3 \cdot tid_{inner} + 2] =$**PADD**$(smem[3 \cdot tid_{inner} + 2], smem[3 \cdot tid_{inner}])$
    end if
    $buffer[offset + num] = smem[3 \cdot tid_{inner} + 2])$
    $buffer\_index[offset + num] = \lfloor \frac{pre\_bucket\_idx+1}{2} \rfloor$
    $smem[3 \cdot tid_{inner} + 2] = \mathcal{O}$, $num = num + 1$, $parity = 0$
  end if
  $pre\_bucket\_idx = \hat{a}_i.ODD$
  $smem[3 \cdot tid_{inner} + parity] = 2^{\hat{a}_i.EXP} P_{p_i}$
  if $\hat{a}_i.SIGN = 1$ **then**
    $smem[3 \cdot tid_{inner} + parity] = -smem[3 \cdot tid_{inner} + parity]$
  end if
  $parity = parity \oplus 1$
  if $parity = 0$ **then**
    $smem[3 \cdot tid_{inner} + 1] =$**PADD**$(smem[3 \cdot tid_{inner} + 1], smem[3 \cdot tid_{inner}])$
    $smem[3 \cdot tid_{inner} + 2] =$**PADD**$(smem[3 \cdot tid_{inner} + 2], smem[3 \cdot tid_{inner} + 1])$
  end if
end for
if $parity \neq 0$ **then**
  $smem[3 \cdot tid_{inner} + 2] =$**PADD**$(smem[3 \cdot tid_{inner} + 2], smem[3 \cdot tid_{inner}])$
end if
$buffer[offset + num] = smem[3 \cdot tid_{inner} + 2])$
$buffer\_index[offset + num] = \lfloor \frac{pre\_bucket\_idx+1}{2} \rfloor$
$buffer\_used[tid] = num + 1$

shown in Alg. 5 to complete the entire bucket accumulation phase.

For each subtask, we allocate 2^{c-2} threads, where the thread with index *tid* is used to search for all buffered points corresponding to bucket $B_{2\cdot tid+1}$. The accumulation of these points is done through the shared memory $smem[NTHREADS]$. The worst-case complexity of binary search for the corresponding buffered point is $O(\log N)$, and all subtasks can concurrently run with $\lceil \frac{\lambda}{c} \rceil \cdot 2^{c-2}$ threads.

3.4 A layered parallel reduction algorithm for the bucket aggregation

After obtaining the values of $\{B_t\}$ where $1 \leq t < 2^{c-1}$ and t is an odd integer, we need to calculate the result of *bucket aggregation phase*. Since the number of buckets in our proposed algorithm has been reduced to $\frac{1}{4}$ of that in the original Pippenger algorithm, we no longer calculate the result of Eq. (1). Instead, we compute

$$Q_j = \sum_{i=0}^{2^{c-2}-1} (2 \cdot i + 1) B_{2\cdot i+1} \quad (j = 0, 1, \cdots, c-1)$$

for each subtask. But it is still necessary to update all the buckets $\{B_{2\cdot i+1}\}$ to $\{\hat{B}_{2\cdot i+1} = \sum_{j=i}^{2^{c-2}-1} B_{2\cdot j+1}\}$. For this reason, we propose a parallel layered reduction algorithm for this process that was originally performed sequentially. For clarity, Figure 6 provides an example with only four buckets, which is divided into two rounds: In the first round, we concurrently update the bucket B_5 to $B'_5 = B_5 + B_7$ and the bucket B_1 to $B'_1 = B_1 + B_3$. In the second round, we concurrently update the bucket B'_1 to $\hat{B}_1 = B'_1 + B'_5$ and B'_3 to $\hat{B}_3 = B'_3 + B'_5$. When the number of buckets expands to 2^{c-2} in our scheme, the rounds will also expand to $c-2$ rounds, i.e., $\log_2(2^{c-2})$. In the best case, we can assign 2^{c-3} threads to each subtask with thread indices $tid \in [0, 2^{c-3})$. Then in the i-th round, each thread computes $baseline = 2^i \cdot \lfloor \frac{tid}{2^{i-1}} \rfloor + 2^{i-1}$ and $offset = (tid \% 2^{i-1}) + 1$, and performs a single addition operation:

$$B_{2^{c-1}+1-2\cdot(baseline+offset)} = \mathbf{PADD}(B_{2^{c-1}+1-2\cdot(baseline+offset)}, B_{2^{c-1}+1-2\cdot baseline}).$$

It means that we update the $(baseline + offset)$-th-to-last bucket to be the sum of itself and the $(baseline)$-th-to-last bucket. Even if the total number of threads is not large enough, we can evenly distribute these small tasks that involve only one addition among these threads, ensuring that the computational cost for each thread remains quite low. Moreover, it is not necessary to launch a new kernel for each round but directly use CUDA's *cooperative groups* to achieve thread synchronization across the entire grid. Therefore, the time complexity of our layered parallel reduction scheme remains at the logarithmic level of the number of buckets.

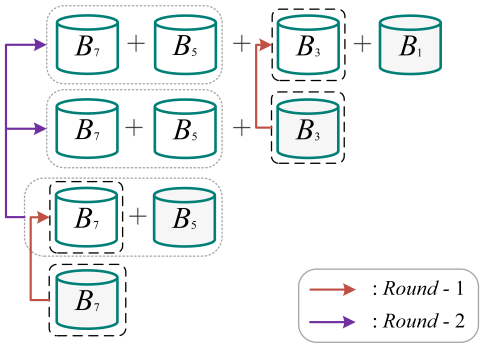

Figure 6: Example of the layered reduction algorithm (scale=4).

Algorithm 5 Aggregation of buffered points into buckets

Input: Four static arrays: $buffer[N + 2^{c-2}]$, $buffer\_offset[N]$, $buffer\_index[N + 2^{c-2}]$, $buffer\_used[N]$ (for a single subtask), and $tid \in [0, 2^{c-2})$
Output: Buckets $\{B_t\}$ $(1 \leq t < 2^{c-1} \wedge t\%2 = 1)$

$left = 0$
$right = N - 1$
$notInf = false$
while $left \leq right$ **do**
  $mid = left + ((right - left) \gg 1)$
  $used = buffer\_used[mid]$
  if $used = 0$ **then**
    $right = mid - 1$
  else
    $offset = buffer\_offset[mid]$
    $idx_{min} = buffer\_index[offset]$
    $idx_{max} = buffer\_index[offset + used - 1]$
    if $idx_{min} = tid + 1$ **then**
      $startPos = mid$
      $notInf = true$
      $right = mid - 1$
    else if $idx_{min} > tid + 1$ **then**
      $right = mid - 1$
    else if $idx_{max} < tid + 1$ **then**
      $left = mid + 1$
    else
      for $i = offset + 1$ **to** $offset + used - 1$ **by** 1 **do**
        if $buffer\_index[i] = tid + 1$ **then**
          $startPos = mid$
          $notInf = true$
          break
        end if
      end for
    end if
  end if
end while
$smem[tid_{inner}] = \mathcal{O}$ ▷ tid_{inner}: intra-block thread index
while $notInf = true$ **do**
  $notInf = false$
  $used = buffer\_used[startPos]$
  $offset = buffer\_offset[startPos]$
  for $i = offset$ **to** $offset + used - 1$ **by** 1 **do**
    if $buffer\_index[i] = tid + 1$ **then**
      $notInf = true$
      $smem[tid_{inner}] =$**PADD**$(smem[tid_{inner}], buffer[i])$
      break
    end if
  end for
  $startPos++$
end while
$B_{2 \cdot tid+1} = smem[tid_{inner}]$

After updating all the buckets using the above method, with $\hat{B}_{2\cdot i+1} = \sum_{j=i}^{2^{c-2}-1} B_{2\cdot j+1}$, we only need to accumulate all these new bucket points to obtain $Q_j = \sum_{i=0}^{2^{c-2}-1} \hat{B}_{2\cdot i+1} = \sum_{i=0}^{2^{c-2}-1}(i+1)B_{2\cdot i+1}$. This part already has mature CUDA parallel implementations. Before the accumulation, we need to record the original $\hat{B}_1$ (i.e., $\sum_{j=0}^{2^{c-2}-1} B_{2\cdot j+1}$), which is used to compute the final result of the subtask: $Q_j = 2Q_j - \hat{B}_1$. Finally we can aggregate the results of $\lceil \frac{\lambda}{c} \rceil$ subtasks $\{Q_j\}$ into the result of the total MSM task.

3.5 Cost analysis

In this subsection, we discuss the computational cost for each phase within each thread.

- Relying on the precomputation tables, the scalar preprocessing requires n/N table lookups. Radix sort is also a stable sorting algorithm, and we use the state-of-the-art implementation from the CUB library.
- The bottleneck of the GPU implementation based on the Pippenger algorithm lies in the bucket accumulation phase. As mentioned in Section 3.3, we divide the bucket accumulation phase into the following two phases:

  (1) Accumulate all points into non-conflicting buffers assigned to each thread.

  (2) Then aggregate the buffered points back into their respective unique buckets.

  Due to the large size of n and the random sampling of scalars, the probability of performing doubling operations in phase(1) can be directly viewed as $1/2^{\tau+1}$. Therefore, the phase(1) requires n/N point additions and a relatively small number of doublings (related to the parameter τ), making the cost quite stable. It should be noted that the above discussion is specific to the algorithm based on cached Jacobian coordinates. When $\tau = c-1$, no doubling operations are needed, although it requires more GPU memory.

  For phase(2), due to hardware constraints, N must be less than the number of buckets. It means that the occurrence of multiple consecutive threads processing the same bucket index in phase(1) is extremely low, i.e., there are very few buffered points that correspond to the same bucket. Thus, each thread in phase(2) requires a negligible number of point additions (usually only once).

- The bucket aggregation phase requires $\log_2(2^{c-2})$ point additions.

However, in the most extreme input scenarios, our bucket accumulation phase is not "constant-time", where "constant-time" means the running-time is independent from the input. For instance, if all scalars are identical, different scalar values are likely to result in different numbers of doubling operations in phase(1), although the "load-balanced" property can still be guaranteed. In addition, all buffered points correspond to the same bucket, requiring the execution of up to N point additions in phase(2). For this, we design a new algorithm to ensure the constant-time property of the bucket accumulation phase. The details of the new algorithm and experimental results are presented in Section 5. Although it is well known that there will be a slowdown when patching the constant time, our experimental results show that the slowdown caused by the new algorithm is entirely within an acceptable range.

4 Implementation Results

In this section, we present the implementation results of our GPU-accelerated MSM algorithm, and compare it with the most recent GPU implementation, namely `cuZK` [LWY+23].

We perform the experiments on three different GPUs: 1) V100, 2) RTX3090, and 3) RTX4090. Detailed hardware information including the CPU configuration on the host side is listed in Table 2. Since the main body of our MSM implementation is done on the GPU, the CPU's performance only affects the scalar transfer time between the host and the device. To address this, we employ multi-stream techniques to overlap data transfers with device computations, further reducing the latency overhead caused by transfers.

Table 2: Hardware configuration of testing environments

Environment	V100	RTX3090	RTX4090
Device	V100-SXM2-32GB	GeForce RTX3090	GeForce RTX4090
SM Count	80	82	128
Core Count	5120	10496	16384
Host(CPU)	Xeon(R) Platinum 8255C	Xeon(R) Platinum 8358P	Xeon(R) Platinum 8352V
CPU Cores	12	15	12
CPU Freq.	2.50GHz	2.60GHz	2.10GHz
OS	Ubuntu 20.04	Ubuntu 20.04	Ubuntu 20.04
CUDA Version	11.3	11.3	11.8

The BLS12-377, BLS12-381, and BLS24-315 curves that we employ are defined over prime fields $\mathbb{F}_{p_1}$, $\mathbb{F}_{p_2}$, and $\mathbb{F}_{p_3}$, respectively. The primes are 377-bit, 381-bit and 315-bit, respectively. Since we store elements from $\mathbb{F}_{p_1}$ (or $\mathbb{F}_{p_2}$) in multiple 32-bit unsigned integers, a single finite field element on both BLS12-377 and BLS12-381 is represented using 384 bits. Similarly, a single finite field element on BLS24-315 can be stored with fewer 320 bits.

In terms of parameter selection, we set the window size c as 16, which can avoid explicit sub-scalar segmentation by the pointer of type `uint16_t`. We set the τ as 6 for the cached Jacobian coordinate-based method and evaluate it on the BLS12-377 and BLS12-381 curves. For example, considering our tested MSM upper limit size of 2^{24}, this requires a memory space of $10.5GB$ to store $7 \cdot n$ affine points (including the base points themselves), which is well within the capabilities of mainstream GPU devices. This parameter selection also ensures that the cases where doubling operations are required during the bucket accumulation phase account for only about $\frac{1}{128}$ of the overall workload, effectively leveraging the precomputation table. For the method based on the homogeneous coordinate system, we generate a complete precomputation table (i.e., $\tau = 15$) and also evaluate it on the SNARK-Friendly BLS24-315 curve. When the MSM size reaches 2^{23}, it requires $10GB$ of memory space to store $16 \cdot n$ affine points (each point occupying $320 \cdot 2$ bits), which is still within an acceptable range.

The shared memory limit is 48KB, and it is at the block level for CUDA. It only launches a block when there is enough free memory to accommodate the entire size required by that block. We set the block size $NTHREAD$ to 64, then the method based on cached Jacobian coordinates occupies $24KB$ of shared memory per block on the BLS12-381 curve, achieving full occupancy ($2 \cdot 24KB$). The method based on homogeneous coordinates occupies $15KB$ per block on the BLS24-315 curve, achieving a 93.75% occupancy ($3 \cdot 15KB$). In addition, we determine the maximum number of threads allocated based on the current maximum number of Streaming Multiprocessors (SMs) in the GPU, specifically, it is $(256 \cdot SM\_Count)$.

ZPrize [Zpr22] is an annual competition with a primary focus on promoting the utilization and advancement of zero-knowledge cryptography. One of the tracks it establishes focuses on accelerating MSM computations using GPUs, and our implementation is based on their testing framework. `The source code is available at` https://github.com/dunkirkturbo/wlc_msm.

Since the number of `uint32_t` used for element storage on both BLS12-377 and BLS12-381 curves is the same, the computational cost for operations like point addition is nearly

identical (as confirmed in our practical testing). Table 3 presents our benchmark results of MSM on the BLS12-381 curve evaluated on different GPUs.

Table 3: Execution times (millisecond) of BLS12-381 MSM on different GPUs (V100/RTX3090/RTX4090) and speedup ratios compared to the recent implementation.

Size	V100		RTX3090		RTX4090	
	cuZK	ours	cuZK	ours	cuZK	ours
2^{19}	115.39	44.97 (2.566×)	69.17	29.89 (2.314×)	51.18	17.95 (2.85×)
2^{20}	195.94	84.28 (2.325×)	112.37	56.91 (1.974×)	77.43	32.86 (2.36×)
2^{21}	321.92	161.08 (1.998×)	183.02	110.82 (1.652×)	113.94	62.92 (1.81×)
2^{22}	574.47	315.51 (1.821×)	326.13	214.94 (1.517×)	185.33	124.21 (1.49×)
2^{23}	1128.36	620.74 (1.818×)	645.15	425.78 (1.515×)	355.22	250.68 (1.42×)
2^{24}	2022.47	1233.87 (1.639×)	1181.98	843.18 (1.402×)	1385.76	500.07 (2.77×)

- Our evaluation primarily focuses on MSM calculations in the $\mathbb{G}_1$ group. The benchmark results indicate that our MSM achieves approximately $63.9\% - 156.6\%$, $40.2\% - 131.4\%$, and $42\% - 185\%$ faster performance than `cuZK` on V100, RTX3090, and RTX4090, respectively (at scales ranging from 2^{19} to 2^{24}). This implies that our approach can be applied to the vast majority of MSM computation scales required by zk-SNARKs and significantly outperforms `cuZK` in terms of performance.
- Our load-balancing method requires a buffer size of $N_{max} + \lceil \frac{\lambda}{16} \rceil \cdot 2^{14}$ cached Jacobian points. Taking RTX3090 as an example, we set the maximum threads N_{max} to $82 \cdot 256$, and therefore, the buffer size mentioned above does not exceed $52MB$ (independent of the MSM size). Additionally, due to the similarity in the number of SMs between V100 and RTX3090, we compare the acceleration on RTX3090 and RTX4090. `cuZK` is a load-balanced solution that utilizes sparse matrices, and the results show that our relative speedup compared to `cuZK` remains largely consistent on RTX4090 as it does on RTX3090. Hence, our performance also exhibits nearly linear growth with an increase in the total allocated threads.

To demonstrate the speedup of each optimization technique employed in Table 3, we first use only our method for mapping scalars (in Section 3.1) and the parallel layered reduction algorithm (in Section 3.4), comparing performance with `cuZK`. After incorporating the lazy reduction technique, we conduct another round of testing. Finally, with the application of the load-balancing algorithm (in Section 3.3), we get the complete benchmark results presented in Table 3. The speedup ratio for each round of the testing mentioned above is shown in Table 4.

- Even in the `"imbal"` version, our implementation achieves up to 2.35×, 2.1×, and 2.3× speedup compared to `cuZK` on V100, RTX3090, and RTX4090, respectively (at scales ranging from 2^{19} to 2^{24}). After adding the lazy reduction technique, the `"imbal+lazy"` version achieves a further acceleration of up to 5.76% compared to the `"imbal"` version. Finally, the complete `"bal+lazy"` version is up to 21.48% faster than the `"imbal+lazy"` version. It should be noted that the `"bal+lazy"` version performs significantly faster on higher computational power devices compared to the `"imbal+lazy"` version.

Table 4: The speedup ratio compared to `cuZK` after adding our optimization techniques one by one (For simplicity, `"imbal"` means we only use the method for mapping scalars and the parallel layered reduction algorithm, `"imbal+lazy"` means we incorporate the lazy reduction technique, and `"bal+lazy"` means we further utilize the load-balancing algorithm).

Size	V100			RTX3090			RTX4090		
	`imbal`	`imbal +lazy`	`bal +lazy`	`imbal`	`imbal +lazy`	`bal +lazy`	`imbal`	`imbal +lazy`	`bal +lazy`
2^{19}	2.35×	2.45×	2.57×	2.10×	2.18×	2.31×	2.30×	2.39×	2.85×
2^{20}	2.12×	2.20×	2.32×	1.81×	1.87×	1.97×	1.89×	1.97×	2.36×
2^{21}	1.84×	1.92×	2.00×	1.52×	1.58×	1.65×	1.42×	1.49×	1.81×
2^{22}	1.69×	1.77×	1.82×	1.39×	1.47×	1.52×	1.18×	1.23×	1.49×
2^{23}	1.70×	1.77×	1.82×	1.42×	1.50×	1.52×	1.13×	1.18×	1.42×
2^{24}	1.54×	1.60×	1.64×	1.32×	1.39×	1.40×	2.22×	2.30×	2.77×

Furthermore, we also evaluate the performance improvement brought about by the load-balancing algorithm on the BLS24-315 curve. Before this, we switch to the homogeneous coordinate on the basis of the `"imbal+lazy"` version, which yields a further performance improvement of approximately 3% (The reason for the improvement not reaching the theoretical value is due to issues such as bank conflicts in the usage of shared memory in CUDA). This gives us a new control group with load imbalance. The comparative results between the imbalanced and balanced versions are shown in Figure 7.[3]

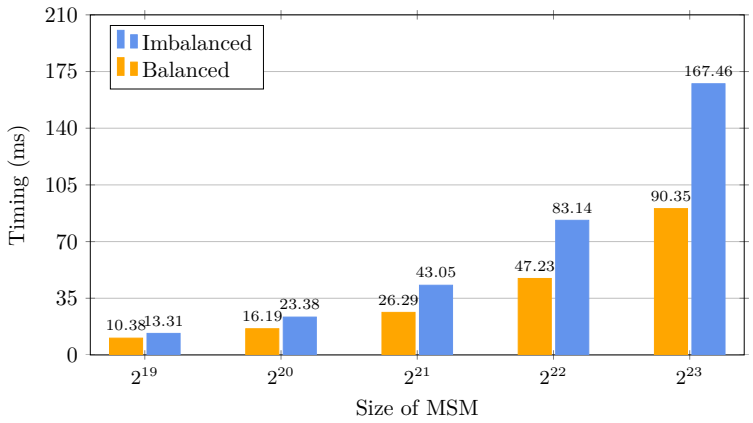

Figure 7: Comparison of the load-balanced and imbalanced versions of our method based on the homogeneous coordinate system (BLS24-315 MSM on RTX4090).

In the load-imbalanced version, we additionally sort the key $\hat{a}_i[0:30]$ during radix sort. Since each thread is responsible for sub-scalar values corresponding to non-overlapping buckets, warp synchronization is achieved for the operation of determining whether the y-coordinate should be negated. Warp synchronization is a critical factor affecting the efficiency of CUDA programs. However, despite this, our load-balanced version is still 28.2% to 85.3% faster than the imbalanced version. Furthermore, as the MSM size gradually increases, this advantage becomes even more pronounced.

In summary, our implementation on SNARK-friendly curves like BLS12-381 and BLS24-315 significantly improves MSM calculation speeds in $\mathbb{G}_1$ compared to the latest implementations. Additionally, it satisfies load balancing and adapts well to GPUs with varying computational power.

[3]Since `cuZK` has not been implemented on the BLS24-315 curve, we do not have a corresponding comparison with their work.

5 Constant-time algorithm for the bucket accumulation

This section reviews the cost analysis from Section 3.5, where the bucket accumulation phase based on the cached Jacobian coordinate is not constant time (here, "constant time" means the running time is independent of the secret input). In particular, for accumulating parts of the buckets into buffers, we can set the parameter τ to $c-1$. It eliminates all doubling operations and the cost per individual thread is fixed at n/N point additions. For aggregating the buffered points into buckets, we design a new algorithm that can be executed in constant time.

For each subtask, the size of the static buffer is $N + 2^{c-2}$. Compared to the previous Alg. 5, we retain only one auxiliary array: *buffer_index*. This array is used to store the bucket offsets corresponding to the points being written. According to our definition of "inter-boundary" in Figure 5, each occurrence of such boundary results in a redundant point. Excluding the bucket offsets corresponding to the redundant points, the array *buffer_index* is non-decreasing. Therefore, we can perform operations OP_1, OP_2, OP_3, and OP_4 in each thread based on the four consecutive *buffer_index* values to be processed (as shown in Table 5).

Table 5: Operations ($OP_1 \to OP_2 \to OP_3 \to OP_4$) corresponding to the relationships among four values in the *buffer_index*. We write A, B, C, D for the points corresponding to these values. The operation **OUT** requires copying a point once.

Relationships	OP_1	OP_2	OP_3	OP_4
$<,<,<$	$A = \mathbf{PADD}(A, \mathcal{O})$	$\mathbf{OUT}(B)$	$\mathbf{OUT}(C)$	$C = \mathbf{PADD}(D, \mathcal{O})$
$<,<,=$	$A = \mathbf{PADD}(A, \mathcal{O})$	$\mathbf{OUT}(B)$	$\mathbf{OUT}(\mathcal{O})$	$C = \mathbf{PADD}(C, D)$
$<,=,<$	$B = \mathbf{PADD}(B, C)$	$\mathbf{OUT}(B)$	$\mathbf{OUT}(\mathcal{O})$	$C = \mathbf{PADD}(D, \mathcal{O})$
$<,=,=$	$C = \mathbf{PADD}(C, B)$	$\mathbf{OUT}(\mathcal{O})$	$\mathbf{OUT}(\mathcal{O})$	$C = \mathbf{PADD}(C, D)$
$=,<,<$	$A = \mathbf{PADD}(A, B)$	$\mathbf{OUT}(C)$	$\mathbf{OUT}(\mathcal{O})$	$C = \mathbf{PADD}(D, \mathcal{O})$
$=,<,=$	$A = \mathbf{PADD}(A, B)$	$\mathbf{OUT}(\mathcal{O})$	$\mathbf{OUT}(\mathcal{O})$	$C = \mathbf{PADD}(C, D)$
$=,=,<$	$A = \mathbf{PADD}(A, C)$	$\mathbf{OUT}(D) \to C$	$\mathbf{OUT}(\mathcal{O})$	$A = \mathbf{PADD}(A, B)$
$=,=,=$	$A = \mathbf{PADD}(A, B)$	$\mathbf{OUT}(\mathcal{O})$	$\mathbf{OUT}(\mathcal{O})$	$C = \mathbf{PADD}(C, D)$

Before accumulating parts of the buckets, it should be noted that we initialize the *buffer* as $\{\mathcal{O}\}$ and the *buffer_index* as $\{0\}$. Then we increment the bucket offsets corresponding to non-redundant points by 1, i.e., add a redundant bucket at the front position. The offset of this redundant bucket is 0 and corresponds to redundant points. After generating the buffered points, due to the attribute of "inter-boundary", the value before the element 0 must be less than the value after it in *buffer_index*. Thus, when determining which relation is in Table 5, the element 0 needs to be equal to the preceding element and smaller than the following element.

It is obvious that each thread requires at most 2 **PADD** and 2 **OUT** operations. Although some relationships may not require the complete execution of these operations, we still use operations such as adding $\mathcal{O}$ to ensure the constant-time property. More importantly, we avoid program branches through techniques such as address offsetting and table lookup. In details, we unify the operations as follows:

$$
\begin{aligned}
&OP_1 : DST_1 = \mathbf{PADD}(SRC_1, SRC_2),\\
&OP_2 : \mathbf{OUT}(SRC_3) \to DST_2,\\
&OP_3 : \mathbf{OUT}(SRC_4) \to DST_3,\\
&OP_4 : DST_4 = \mathbf{PADD}(SRC_5, SRC_6)
\end{aligned}
$$

We encode the less-than relationship as bit 0 and the equal relationship as bit 1. It allows us to precompute a table *op_index* of size 8, where the elements are encoded as $(\delta_1 \ll 28) \mid (\delta_2 \ll 24) \mid (\delta_3 \ll 20) \mid (\delta_4 \ll 16) \mid (\delta_5 \ll 12) \mid (\delta_6 \ll 8) \mid (\delta_7 \ll 4) \mid (\delta_8)$. Then

we allocate two arrays of length 6 in each thread, aux_1 and aux_2. The former stores the addresses of points A, B, C, D, followed by the addresses of $\mathcal{O}$ and a copy of the address of point D. The latter stores the addresses of the buckets corresponding to points A, B, C, D, followed by the addresses of $\mathcal{O}$ and a copy of the address of point C. Assuming the corresponding value of the relationship being processed by the current thread is i, the operations can be rewritten as follows:

$$\begin{aligned}
&OP_1 : aux_1[\delta_1] = \mathbf{PADD}(aux_1[\delta_2], aux_1[\delta_3]),\\
&OP_2 : \mathbf{OUT}(aux_1[\delta_4]) \rightarrow aux_2[\delta_4],\\
&OP_3 : \mathbf{OUT}(aux_1[\delta_5]) \rightarrow aux_2[\delta_5],\\
&OP_4 : aux_1[\delta_6] = \mathbf{PADD}(aux_1[\delta_7], aux_1[\delta_8])
\end{aligned}$$

The required operand addresses can be obtained through the same table lookup in all branches. Then the exactly same operations $OP_1 \rightarrow OP_2 \rightarrow OP_3 \rightarrow OP_4$ are executed. After that, only the new points A and C are useful. However, we still need to update the corresponding values in *buffer_index*. Similarly, we allocate another array, aux_3, with a length of 5. It stores the bucket offsets corresponding to points A, B, C, D, followed by a zero. Then we perform operations OP_5, OP_6 and OP_7, where OP_6 is designed for the " $=, =, <$ " case. It should be clarified that these operations are still performed on each branch.

$$\begin{aligned}
&OP_5 : aux_3[\delta_1] = aux_3[\delta_2] \mid aux_3[\delta_3],\\
&OP_6 : aux_3[2] = aux_3[2] + (aux_3[3] - aux_3[2]) \cdot (\delta_4 == 5),\\
&OP_7 : aux_3[\delta_6] = aux_3[\delta_7] \mid aux_3[\delta_8]
\end{aligned}$$

Therefore, the complete set of operations includes OP_1~OP_7. After each round of parallel processing, the buffered points and corresponding bucket offsets to be processed will be reduced by half, and the adjacent distances will also double. We can complete the aggregation of buffered points into buckets within $\log_2(N + 2^{c-2}) - 1$ rounds. The operations in the final round require two more additions to handle the remaining two points. In summary, our new algorithm for aggregating buffered points requires $2\log_2(N + 2^{c-2})$ point additions and $2(\log_2(N + 2^{c-2}) - 1)$ copies within each thread. OP_5~OP_7 only involve bitwise operations with relatively low overhead. All of the above operations are independent of the values of input scalars. We have theoretically achieved the constant-time property in the bucket accumulation phase.

It should be noted that if all scalars are identical, there will only be N useful buffered points (i.e., there are 2^{c-2} points at infinity). This will result in many instances of OP_1 and OP_4 containing $\mathcal{O}$ in the operands. After aggregating the buffered points into buckets, there are also $2^{c-2} - 1$ buckets equal to $\mathcal{O}$. The popular implementation of **PADD** usually starts by checking whether there is a $\mathcal{O}$ in the operands. If $\mathcal{O}$ is found, it directly outputs the other operand. Therefore, if we apply this type of implementation, the overall running time of the corner case (i.e., all scalars are identical) is less than that of the random instance. To solve this problem, we also patch the **PADD** operation. We first execute addition formulas consistently, and finally check whether there is a $\mathcal{O}$ in the operands. If it does not exist, we copy the temporary result to the output. Otherwise, we copy the other operand. The copy cost of these two cases is exactly the same.

Experimental results. We do benchmark tests on our new algorithm with different cases, including corner and average scalar distribution. In the average case, all n scalars are randomly sampled in $\mathbb{F}_r$, where r is the order of $\mathbb{G}$. While in the corner case, scalars are still random but identical. Our experiments correspond with the average of 1000 executions. The results on V100 are shown in Table 6.

Since the selection of the parameter τ directly determines the cost of doublings when generating buffered points, increasing τ to 15 eliminates all doubling operations. It can

Table 6: Execution times (millisecond) of BLS12-381 MSM on V100 (based on the cached Jacobian coordinate). For simplicity, `"const"` denotes satisfying the constant-time property, while `"non-constant"` does not. `"avg"` means scalars are randomly sampled, and `"corner"` means all scalar values are random but identical.

Size	cuZK	$\tau = 6$ non-const (avg)	$\tau = 15$ non-const (avg)	$\tau = 15$ const (avg)	$\tau = 15$ non-const (corner)	$\tau = 15$ const (corner)
2^{19}	115.39	44.97	19.75	**21.70**	39.36	**21.59**
2^{20}	195.94	84.28	33.89	**36.33**	52.80	**36.42**
2^{21}	321.92	161.08	60.93	**62.22**	82.45	**62.49**
2^{22}	574.47	315.51	116.84	**117.69**	143.59	**116.65**
2^{23}	1128.36	620.74	227.31	**217.74**	243.38	**218.21**

be observed that the performance has been significantly improved, but requiring more GPU memory. For instance, the MSM with a size of 2^{23} requires precomputing $16 \cdot 2^{23}$ affine points (including the base points themselves). It occupies $12GB$ of memory, but when $\tau = 6$, it only requires $5.25GB$. This increase in memory usage remains within an acceptable range and is applicable to popular GPUs, such as the V100-SXM2-32GB.

Then if we apply the new algorithm for aggregating the buffered points into buckets, we will obtain a constant-time version. In the average case, the performance of the constant-time version is close to that of the non-constant-time one. Although more additions are required to aggregate buffered points compared to Alg. 5, the warp synchronization also contributes to the faster parallel execution of several operations such as **PADD**.

If the parameter τ is fixed at 15, the non-constant-time version will experience a performance drop in the corner case. As analyzed in Section 3.5, it requires at most N point additions to aggregate buffered points in one of the threads. However, the performance drop in the corner case becomes less noticeable with an increase in n, as N is relatively smaller than n. For example, the performance decreases by 50.2% when $n = 2^{19}$, while only decreases by 6.8% when $n = 2^{23}$. In contrast, the running time of our constant-time version in the corner case is almost indistinguishable from the average case. For example, the running time differs by less than 0.22% when $n = 2^{23}$.

The slight difference in running time of the constant-time version is due to CUDA's internal optimizations for memory access that we cannot adjust (e.g., coalesced memory access [NVI23]). When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. And the memory transaction is up to 128 bytes. In the BLS12-381 curve, $|\mathbb{F}_p| = 48$-byte but $|\mathbb{G}| = 192$-byte> 128-byte (based on the cached Jacobian coordinate). Recalling our implementation, different threads do not access points at the same global memory location. Thus, these accesses to points will not be coalesced.

The affected accesses are only to the scalar mapping table and *op_index*, where each element is 4-byte. Global memory accesses are always cached in L2 cache (i.e., using 32-byte memory transactions). Therefore, a memory transaction will cover 8 elements in these two tables.

- For *op_index*: The total length of this array is 32-byte, so these accesses are coalesced regardless of the input.

- For the scalar mapping table: When the elements in the table mapped by sub-scalars within the same warp appear in consecutive 32-byte, the accesses will be coalesced. Assuming the attacker is in an ideal scenario without benchmark errors, if he knows that the running time of n scalars is less than that of another n scalars, he can only

infer that the difference between certain sub-scalars within the same warp does not exceed 7. Even if the small range of 32 sub-scalars whose accesses will be coalesced is given, there are at least $(\frac{32}{4})^{32}$ possibilities. Moreover, the small range is also indeterminable in reality.

Based on the above analysis, along with the inherent benchmark errors, the information leakage introduced by CUDA's internal memory access optimization is negligible.

6 Conclusion

In this work, we present a novel GPU-accelerated MSM algorithm that can be applied to large-size scenarios required by zk-SNARKs and achieves high performance. First, we propose a new method for mapping scalars into Pippenger's bucket indices, reducing the number of buckets to $\frac{1}{4}$ of that in the original Pippenger algorithm. Second, we focus on the fundamental operations of point addition during the bucket accumulation phase. We employ mixed point addition formulas in cached Jacobian coordinates when the GPU memory is of typical size, and switch to a more efficient algorithm based on homogeneous coordinates which can save one finite field multiplication cost for every two additions when memory is sufficient. In addition, we also apply the lazy reduction skill to all point addition and doubling operations. Third, we present our load-balanced bucket accumulation method using the compact static buffer, which means the parallel speedup ratio is almost linear growth as the number of device threads increases. Finally, we provide a parallel layered reduction algorithm for the serially executed bucket aggregation phase, whose time complexity remains at the logarithmic level of the number of buckets.

Utilizing the aforementioned techniques, our MSM achieves approximately 40.2% to 185.1% faster performance than `cuZK` on popular graphics cards (at scales ranging from 2^{19} to 2^{24}). Our approach can also be further optimized at the lower level. Longa [Lon23] proposed an approach that generalizes interleaved modular multiplication algorithms for the computation of sums of products over large prime fields. It can avoid the penalty of double-precision operations and perform faster over the popular SNARK-friendly curve. Andy [AM22] et al. presented a LSD radix sorting algorithm for large GPU sorting problems residing in global memory. Compared to CUB which we use, it provides a speedup of $\approx 1.5\times$. The optimizations using these algorithms are left as future work.

Acknowlegements. We thank the anonymous reviewers for their helpful discussion and feedback. The work was supported by the National Key Research and Development Program of China (No. 2021YFA1000600), the National Natural Science Foundation of China (Nos. 62325209, 62172307, 62272350), and the Major Program(JD) of Hubei Province (No. 2023BAA027).

References

[6bl22] 6block. Z-prize msm on the gpu submission. https://github.com/z-prize/2022-entries/tree/main/open-division/prize1-msm/prize1a-msm-gpu/6block, 2022.

[AKL+11] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 48–68, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[AM22] Andy Adinets and Duane Merrill. Onesweep: A faster least significant digit radix sort for gpus. *arXiv preprint arXiv:2206.01784*, 2022.

[BDLO12] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In Steven Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012*, pages 454–473, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BEH23] Gautam Botrel and Youssef El Housni. Faster montgomery multiplication and multi-scalar-multiplication for snarks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):504–521, Jun. 2023.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 416–432, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[BSCG+14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[CC86] D.V Chudnovsky and G.V Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 51–65, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[EHG22] Youssef El Housni and Aurore Guillevic. Families of snark-friendly 2-chains of elliptic curves. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 367–396, Cham, 2022. Springer International Publishing.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

[KKAK96] C. Kaya Koc, T. Acar, and B.S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[LFG23] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):358–380, Mar. 2023.

[Lon23] Patrick Longa. Efficient algorithms for large prime characteristic fields and their application to bilinear pairings. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):445–472, 2023.

[LWY+23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuzk: Accelerating zero-knowledge proof with

a faster parallel multi-scalar multiplication algorithm on gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):194–220, Jun. 2023.

[Mat22] MatterLab. Accelerating msm operations on gpu/fpga. https://github.com/matter-labs/z-prize-msm-gpu, 2022.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[NVI23] NVIDIA. Cuda c++ programming guild, release 12.3. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2023.

[NZ23] Ning Ni and Yongxin Zhu. Enabling zero knowledge proof by accelerating zk-snark kernels on gpu. *Journal of Parallel and Distributed Computing*, 173:20–31, 2023.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, SFCS '76, page 258–263, USA, 1976. IEEE Computer Society.

[Pow90] David MW Powers. *Parallelized quicksort with optimal speedup*. Universität Kaiserslautern. Fachbereich Informatik. Artificial Intelligence . . . , 1990.

[RWGM23] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zkcreds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 790–808, 2023.

[Sco07] Michael Scott. Implementing cryptographic pairings. In *Proceedings of the First International Conference on Pairing-Based Cryptography*, Pairing'07, page 177–196, Berlin, Heidelberg, 2007. Springer-Verlag.

[Wil76] Louis F. Williams. A modification to the half-interval search (binary search) method. In *Proceedings of the 14th Annual Southeast Regional Conference*, ACM-SE 14, page 95–101, New York, NY, USA, 1976. Association for Computing Machinery.

[Yrr22] Yrrid. Z-prize msm on the gpu submission. https://github.com/yrrid/submission-msm-gpu, 2022.

[Zpr22] Zprize. Accelerating msm operations on gpu. https://www.zprize.io/prizes/accelerating-msm-operations-on-gpu-fpga, 2022.