cuZK: Accelerating Zero-Knowledge Proof with A Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs

Tao Lu[1], Chengkun Wei[1], Ruijing Yu[1], Chaochao Chen[1], Wenjing Fang[2], Lei Wang[2], Zeke Wang[1] and Wenzhi Chen[1]

[1] Zhejiang University, Hangzhou, China,
{lutao2020,weichengkun,rjyu,zjuccc,wangzeke,chenwz}@zju.edu.cn
[2] Ant Group, Hangzhou, China, {bean.fwj,shensi.wl}@antgroup.com

Abstract. Zero-knowledge proof is a critical cryptographic primitive. Its most practical type, called zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK), has been deployed in various privacy-preserving applications such as cryptocurrencies and verifiable machine learning. Unfortunately, zkSNARK like Groth16 has a high overhead on its proof generation step, which consists of several time-consuming operations, including large-scale matrix-vector multiplication (MUL), number-theoretic transform (NTT), and multi-scalar multiplication (MSM). Therefore, this paper presents cuZK, an efficient GPU implementation of zkSNARK with the following three techniques to achieve high performance. First, we propose a new parallel MSM algorithm. This MSM algorithm achieves nearly perfect linear speedup over the Pippenger algorithm, a well-known serial MSM algorithm. Second, we parallelize the MUL operation. Along with our self-designed MSM scheme and well-studied NTT scheme, cuZK achieves the parallelization of all operations in the proof generation step. Third, cuZK reduces the latency overhead caused by CPU-GPU data transfer by 1) reducing redundant data transfer and 2) overlapping data transfer and device computation. The evaluation results show that our MSM module provides over 2.08× (up to 2.94×) speedup versus the state-of-the-art GPU implementation. cuZK achieves over 2.65× (up to 4.86×) speedup on standard benchmarks and 2.18× speedup on a GPU-accelerated cryptocurrency application, Filecoin.

Keywords: Zero-knowledge Proof · Multi-scalar Multiplication · Parallel Algorithm · Graphics Processing Unit

1 Introduction

Zero-knowledge proof (ZKP) [GMR89] is a cryptographic primitive that allows a prover to generate a proof π to convince verifiers that a computation $y = f(x, w)$ is correctly calculated with a public input x and a prover's secret input w. Additionally, the proof π leaks no information about the secret input w. In recent years, ZKP has drawn much attention from academia and industry due to the advent of an advanced ZKP type called zkSNARK, which stands for *zero-knowledge Succinct Non-interactive ARgument of Knowledge*. Compared with other traditional ZKPs [Kil92, Mic00, Gro10], zkSNARK has much more succinct proof π. For example, the proof generated in the zkSNARK protocol proposed by Groth [Gro16] has only hundreds of bytes and is very fast to be verified within several milliseconds. Therefore, zkSNARK is widely considered to be the most practical ZKP, and it has been applied to many private-preserving applications such

as electronic voting [ZC15], verifiable database outsourcing [ZGK+17], cryptocurrencies [SCG+14, BG17, BMRS20], and verifiable machine learning [ZFZS20].

However, there is still a bottleneck that limits further deployments of zkSNARK. Currently, state-of-the-art zkSNARKs have a high overhead on their proof generation step. The prover in [Gro16] needs to perform various time-consuming operations to generate a proof π. These time-consuming operations include large-scaled matrix-vector multiplication (MUL), number-theoretic transform (NTT), and multi-scalar multiplication (MSM) on elliptic curves, leading to the overall proof generation time for a function f being much longer than the time to evaluate this function, sometimes up to thousands of times longer.

One of solutions to reduce the proof generation time is to parallelize this task on certain hardware. GPUs are many-core computing platforms that support the concurrent execution of thousands of threads. They have been used to accelerate a wide variety of computational modules in many fields, such as deep learning [LZW21], cryptography [GXW21, ABVL+20], and graphics [WQS+20]. There are also several existing GPU designs of zkSNARK. For example, Mina announced a challenge for speeding up [Gro16] using GPUs with a high reward ($ 100k). The final acceleration result of this challenge has been open-sourced in [Min19]. Another GPU implementation Bellperson [Bel19] is improved from a CPU-based version Bellman [Bel15]. Bellperson has been deployed in a well-known decentralized cryptocurrency network Filecoin [BG17]. Figure 1 shows their execution time breakdown on zkSNARK operations, including MUL, NTT, MSM, and the GPU-CPU data transfer. Obviously, the overall performance of these GPU implementations largely depends on the efficiency of the above four operations. Especially, MSM is the most time-consuming operation, taking more than 70 percent of the total runtime.

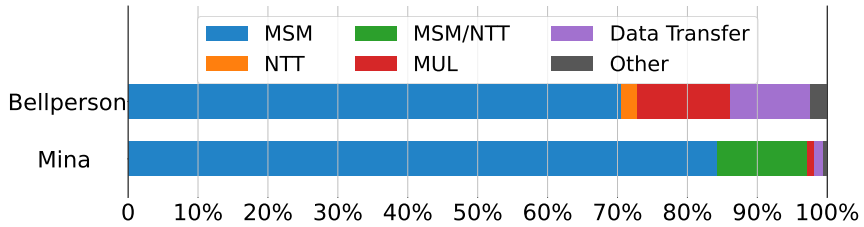

Figure 1: The execution time breakdown of the existing GPU implementations on zkSNARK operations, including MUL, NTT, MSM, and the GPU-CPU data transfer. The label MSM/NTT means that MSM and NTT are executed simultaneously.

Nevertheless, these operations performed in existing GPU implementations have the following weaknesses. **1) MSM:** The parallel algorithms used in these GPU implementations for the MSM computation are simply modified from the ones used in the low-parallelism setting. However, these parallel algorithms are hardly suitable for the case when there are thousands of threads running simultaneously, which manifests an unsatisfiable increase in speedup with the increasing parallelism; **2) MUL:** We notice existing GPU designs underestimate computational costs of the MUL operation. They choose to perform the MUL operation serially on a CPU rather than parallelly on GPUs. Actually, the slow way of running MUL serially can significantly hinder the overall performance; **3) Data Transfer:** These GPU implementations also waste too much time on CPU-GPU data transfer, which can actually be mitigated by reducing redundant data transfer and overlapping data transfer with device computation. Note that Mina [Min19] consumes much time in performing NTT serially, but there is no need to consider NTT to be a weakness. Many efficient parallel NTT schemes already have been well-studied [GJCC20, KJPA20, GXW21]. Therefore, we can easily replace the serial NTT scheme used in [Min19] with a parallel one. For example, the parallel NTT scheme used in [Bel19] is from [Bai10].

1.1 Our Contributions

In this paper, we provide an efficient GPU implementation of zkSNARK by addressing the above weaknesses of other state-of-the-art works. Proposed techniques can achieve high performance on modern GPU architectures. Contributions of this paper are summarized below:

- We propose a new parallel MSM algorithm. This MSM algorithm is unlike other ordinary parallel methods that simply decompose the large MSM computation into multiple smaller ones. We treat all computational units of MSM as a whole and store all elements of MSM in a sparse matrix. This enables us to convert the major operations used in the Pippenger algorithm [Pip76], a well-known serial MSM algorithm, to a series of basic sparse matrix operations, including sparse matrix transpose and sparse matrix-vector multiplication. Next, we utilize the technologies used in well-studied parallel sparse matrix algorithms [BG09, GD14, TDM+14] to accelerate the MSM computation. As a result, our parallel MSM algorithm is not only well adapted to the high parallelism provided by GPUs, but also achieves nearly perfect linear speedup over the Pippenger algorithm, where perfect linear speedup means the parallel speedup ratio is equal to the number of execution threads. We are the first to show that the MSM computation can be parallelized with the help of sparse matrix operations, and we believe that this idea of using sparse matrices will motivate many other parallelization methods due to its excellent performance.

- We present cuZK, an efficient GPU implementation of zkSNARK. We make three optimizations to help cuZK achieve high performance. First, we implement our new parallel MSM algorithm and deploy it in cuZK's MSM module. This part dominates the total costs of zkSNARK and thus dramatically impacts the overall performance improvement. Second, we notice the matrix that MUL operates on is very large but sparse. Therefore, we represent it in the sparse matrix format, which allows us to store the whole matrix on GPUs and perform MUL computation with parallel schemes on sparse matrices. Furthermore, along with our self-designed MSM parallel scheme and well-studied NTT parallel scheme [GJCC20, KJPA20, GXW21], cuZK indeed achieves the parallelization of all time-consuming operations in zkSNARK. Third, cuZK reduces the latency overhead caused by CPU-GPU data transfer by overlapping data transfer and device computation using the multi-streaming technique. In addition, there is no need for redundant data transfer as it is automatically eliminated when we perform all zkSNARK operations on GPUs.

- We design a series of evaluation schemes for cuZK. The evaluation results show that our MSM module provides over 2.08× (up to 2.94×) speedup versus the state-of-the-art GPU implementation. The overall performance of cuZK achieves over 2.65× (up to 4.86×) speedup on standard benchmarks and 2.18× speedup on a GPU-accelerated cryptocurrency application, Filecoin.

Implementation codes discussed in this paper are available in https://github.com/speakspeak/cuZK.

1.2 Related Work

Recently, a great number of prior works have implemented high-performance zkSNARK on certain hardware, including GPUs, ASICs, and CPU clusters. PipeZK [ZWZ+21] is a work that provides pipelined ASIC design for zkSNARK. Although this work has excellent efficiency on zkSNARK for small-scale applications like anonymous payment, its performance decreases significantly for large-scale applications due to the on-chip

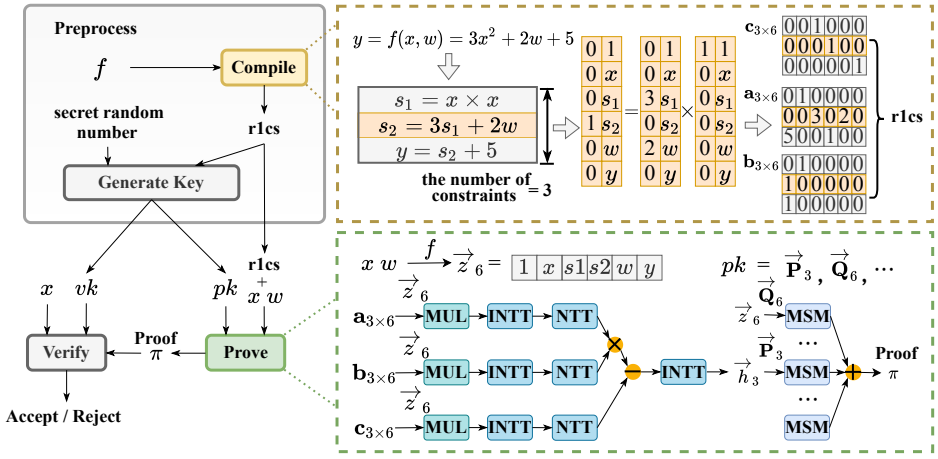

Figure 2: The workflow of Groth's protocol, where INTT is the inverse transformation of NTT. Its operation is similar to NTT.

storage limitation of ASIC. DIZK [WZC+18] leverages Apache Spark to distribute the proof generation step to CPU clusters. Nevertheless, the costs of deploying CPU clusters are much higher than using GPU cards and ASIC chips, which hinders DIZK from being widely deployed. Bellperson [Bel19] and Mina [Min19] are efficient GPU implementations of zkSNARK. However, as mentioned before, their MSM and MUL modules do not fully unleash the potential of GPUs and they also waste too much time on CPU-GPU data transfer.

MSM is the most time-consuming operation used in zkSNARK. Thus, there have been notable works focusing on improving the efficiency of MSM on GPUs [Spp22, Yrr22, Mat22], FPGAs [Xav22, ABC+22, Har22], and ARM CPUs [HB22]. Especially the winning works [Yrr22, Mat22] of the ZPrize competition [Zpr22] are the concurrent works with this paper. They achieve excellent performance on the MSM computations with randomly sampled scalars. Their excellent performance benefits from the core technique, i.e. utilizing radix sort to process the scalars used in MSM, which can actually be viewed as a concrete scheme of the sparse matrix transpose used in our MSM algorithm, as shown in Section 4.1. But even then, both schemes cannot be directly deployed on zkSNARK, where the scalars used in MSM depend on the proving function f and are not simply randomly distributed.

2 Preliminaries

2.1 The zkSNARK Protocol of Groth

Our work provides a GPU implementation of a zkSNARK protocol due to Groth [Gro16]. We choose Groth's protocol because it is one of the most efficient and practical zkSNARK protocols, and it has been adopted by various private-preserving applications, including cryptocurrencies [BG17], smart contracts [Pol22], and verifiable machine learning [ZWW+21]. Here, we also need to mention that our techniques can be used for similar zkSNARK protocols, especially for the protocols [GGPR13, GM17, GWC19, CHM+20] that require performing the multi-scalar multiplication operation. Below, we describe the details of Groth's protocol.

Groth's protocol works like all zkSNARKs. It allows a prover to generate a proof π to convince verifiers a computation $y = f(x, w)$ is correctly calculated with a public input x and a prover's secret input w. In addition, its proof π has only hundreds of bytes and is

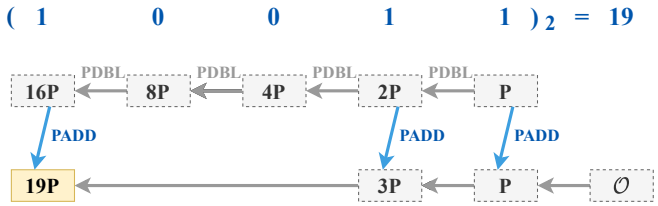

Figure 3: An example of PMUT computation. $\mathcal{O}$ is the point at infinity on elliptic curve.

very fast to be verified within several milliseconds. The workflow of Groth's protocol is shown in Figure 2. It consists of three procedures: `Preprocess`, `Prove`, and `Verify`.

The `Preprocess` procedure is performed by a third trusted party. It first compiles a function f to an instance of Rank-1 Constraint System (R1CS). A simple example of the compilation process is shown on the upper right side of Figure 2.

In short, the function f is decomposed into multiple constraints, each of which can be represented by three vectors. These constraint vectors ultimately form three matrices, which are called the R1CS instance. The number of constraints is commonly considered as the scale of the R1CS instance. Next, the third trusted party uses this R1CS instance and its secret random number to generate a prover key pk and a verifier key vk. These two keys are both public. That is to say, anyone can perform the `Prove` procedure to generate a proof π with the prover key pk, and anyone can perform the `Verify` procedure with the proof π and the verifier key vk. The restriction is that only the proof π generated by the prover who owns the secret input w that satisfies $y = f(x, w)$ can make verifiers accept. In addition, the proof π leaks no information about the secret input w, and no one except the owner can get the value of the secret input w.

The `Preprocess` procedure and the `Verify` procedure have amortized lightweight computational costs. For the `Preprocess` procedure, the two keys pk and vk are infinitely reusable for the function f so that its computational costs can be amortized over each use of two keys. For the `Verify` procedure, it only requires verifiers to perform three bilinear pairing operations like Weil pairing and Tate pairing. These pairing operations are functions that map two mathematical spaces to a third space, and they can be computed using Miller's algorithm [Mil04] within just several milliseconds. The `Prove` procedure is the only high-expense procedure. As shown at the lower right side of Figure 2, it requires the prover to perform various time-consuming operations, including MUL, NTT, and MSM. This leads to the `Prove` procedure being the bottleneck that limits zkSNAKR further deployments. Our work focuses on accelerating this procedure with GPUs.

2.2 Multi-scalar Multiplication

Multi-scalar multiplication (MSM) is the most time-consuming operation in zkSNARK, taking more than 70 percent of the total runtime, as shown in Figure 1. Its definition is given by the formula $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$, where n is the scale of MSM, k_i is a λ-bits scalar, $\mathbf{P}_i$ is an elliptic curve (EC) point, and the pair $k_i\mathbf{P}_i$ represents point scalar multiplication (PMULT) of k_i and $\mathbf{P}_i$. Formally, an elliptic curve is a smooth, projective, algebraic curve consisting of EC points. These points include the set that satisfies a specific mathematical equation, such as $y^2 = x^3 + ax + b$, and the point at infinity, denoted as $\mathcal{O}$. The point at infinity serves as the identity element in the abelian group formed by all EC points, along with their fundamental operation known as point addition (PADD). Point doubling (PDBL) is a special case of PADD, the result of which is equal to performing a PADD operation on two identical points. PMULT of a scalar k and an EC point $\mathbf{P}$ is another commonly used operation in elliptic curve arithmetic. It is defined as k times self-PADD of $\mathbf{P}$, denoted by $k\mathbf{P} = \mathbf{P} + \mathbf{P} + ... + \mathbf{P}$. We can use the double-and-add method to compute

Algorithm 1 The Pippenger Algorithm [Pip76]

Require: A scalar vector $\overrightarrow{k}_n = [k_1, k_2, ..., k_n]$, whose elements are λ-bit scalars. A point vector $\overrightarrow{\mathbf{P}}_n = [\mathbf{P}_1, \mathbf{P}_2, ..., \mathbf{P}_n]$. A chosen window size s.
Ensure: $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$

$\mathbf{T}_{\lceil \frac{\lambda}{s} \rceil + 1} \leftarrow \mathcal{O}$ *// $\mathcal{O}$ is the point at infinity on the elliptic curve.*
for $j \leftarrow (\lceil \frac{\lambda}{s} \rceil)$ to 1 **do** *// Convert the original task into $\lceil \frac{\lambda}{s} \rceil$ subtasks.*
  // Initialize $2^s - 1$ buckets with points at infinity $\mathcal{O}$.
  $\overrightarrow{\mathbf{B}}_{2^s-1} \leftarrow [\mathcal{O}, \mathcal{O}, ..., \mathcal{O}]_{2^s-1}$
  // Put $\mathbf{P}_i$ into the corresponding bucket and add up all points in the same bucket.
  for $i \leftarrow 1$ to n **do**
    // m_{ij} is a part of k_i used in this subtask.
    $m_{ij} \leftarrow (k_i \gg ((j-1) * s))$ & $((1 \ll s) - 1)$
    if $m_{ij} \neq 0$ **then**
      $\mathbf{B}_{m_{ij}} \leftarrow \mathbf{B}_{m_{ij}} + \mathbf{P}_i$
    end if
  end for
  // Get the result of this subtask, $\mathbf{G}_j = \sum_{l=1}^{2^s-1} l\mathbf{B}_l$, as shown in Algorithm 2
  $\mathbf{G}_j \leftarrow \mathsf{BucketPointsReduction}(\overrightarrow{\mathbf{B}}_{2^s-1})$
  $\mathbf{T}_j \leftarrow 2^s \mathbf{T}_{j+1} + \mathbf{G}_j$ *// Add $\mathbf{G}_j$ to the final result based on Formula (3).*
end for
$\mathbf{Q} = \mathbf{T}_1$
return Q

the pair $k\mathbf{P}$ by performing a series of PDBLs and PADDs. Figure 3 shows an example of computing $19\mathbf{P}$. It starts with representing the scalar 19 in the binary form $(10011)_2$. Then, we initialize the result to be the point at infinity $\mathcal{O}$. Next, at each bit position, It doubles and adds the point $\mathbf{P}$ to the result when the bit is 1. The EC point on the last bit is the result of PMULT. Finally, the MSM result $\mathbf{Q}$ is calculated by adding all pairs $k_i\mathbf{P}_i$, where $i \in [1, n]$.

Obviously, if we employ the double-and-add method to compute MSM, we need to perform at most $n\lambda + n - 1$ PADDs and $n\lambda - n$ PDBLs. In real-world applications, the security parameter λ commonly ranges from 254 to 768 and the scale of MSM n could be extremely large. For instance, Filecoin [BG17] has n larger than a million. To make matters worse, the costs of EC point operations like PADD and PDBL are much more expensive than the regular scalar operations. Therefore, the computational costs of using the double-and-add method for MSM computation are intolerable. There are several more efficient MSM algorithms, such as the Pippenger algorithm [Pip76], the Chang-Lou algorithm [CL03], and the Bos-Coster algorithm reported in [Roo94]. Especially, the Pippenger algorithm performs best when the scale of MSM is very large, as shown in [BDLO12].

2.3 The Pippenger Algorithm

The Pippenger algorithm [Pip76] is a popular serial algorithm for MSM computation. It performs best in the zkSNARK setting compared to other MSM algorithms. Our proposed parallel MSM algorithm is inspired by it. Thus, in this section, we first review the Pippenger algorithm and then analyze its computational costs.

Overview. Given λ-bits scalars $k_1, ..., k_n$ and base points $\mathbf{P}_1, ..., \mathbf{P}_n$, the Pippenger algorithm chooses a window size s and converts the original MSM task to $\lceil \frac{\lambda}{s} \rceil$ subtasks by dividing these λ-bits scalars into s-bits scalars. Each subtask is to compute $\sum_{i=1}^{n} m_i \mathbf{P}_i$,

Algorithm 2 BucketPointsReduction [BDLO12]

Require: A point vector $\overrightarrow{\mathbf{B}}_{2^s-1} = [\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{B}_{2^s-1}]$
Ensure: $\mathbf{G} = \sum_{l=1}^{2^s-1} l\mathbf{B}_l$
1: $\mathbf{G}_0 \leftarrow \mathcal{O}$; $\mathbf{M}_0 \leftarrow \mathcal{O}$ // $\mathcal{O}$ *is the point at infinity on the elliptic curve.*
2: **for** $l \leftarrow 1$ to $2^s - 1$ **do** // *Add* $l\mathbf{B}_l$ *based on Formula (2).*
3: $\mathbf{M}_l \leftarrow \mathbf{M}_{l-1} + \mathbf{B}_{2^s-l}$ // $\mathbf{M}_l = \mathbf{B}_{2^s-1} + \mathbf{B}_{2^s-2} + ... + \mathbf{B}_{2^s-l}$
4: $\mathbf{G}_l \leftarrow \mathbf{G}_{l-1} + \mathbf{M}_l$ // $\mathbf{G}_l = \mathbf{M}_1 + \mathbf{M}_2 + ... + \mathbf{M}_l$
5: **end for**
6: $\mathbf{G} \leftarrow \mathbf{G}_{2^s-1}$ // $\mathbf{G}_{2^s-1} = \mathbf{B}_1 + 2\mathbf{B}_2 + ... + (2^s - 1)\mathbf{B}_{2^s-1}$
7: **return G**

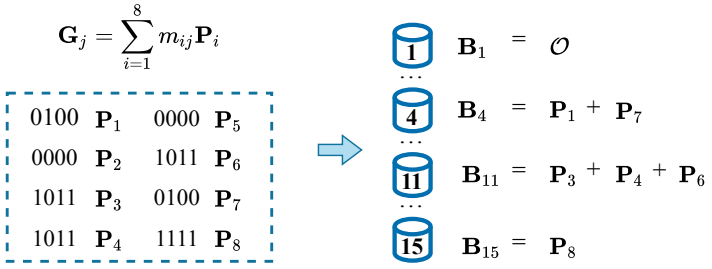

Figure 4: An example of putting EC points into buckets.

where m_i is the s-bits part of k_i used in this subtask. The above step can be performed by sorting base points into $2^s - 1$ buckets according to the value of m_i (discarding one bucket because scalars equal to zero have no effect on the final result). Then, the algorithm sums the base elements in the buckets to obtain points $\mathbf{B}_1, ..., \mathbf{B}_{2^s-1}$ and computes $\sum_{l=1}^{2^s-1} l\mathbf{B}_l$, which is equal to the result of the subtask $\sum_{i=1}^{n} m_i\mathbf{P}_i$.

Details. The details are shown in Algorithm 1, which mainly consists of three steps:

1) Convert the original task $\mathbf{Q} = \sum_{i=1}^{n} k_i\mathbf{P}_i$ to multiple smaller subtasks. It start by choosing a window size s and divides each λ-bits scalar k_i into $\lceil \frac{\lambda}{s} \rceil$ parts. Each part is a s-bits scalar m_{ij}, satisfying $k_i = \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil} (2^{(j-1)s} m_{ij})$. The smaller subtasks are defined as the computation $\mathbf{G}_j = \sum_{i=1}^{n} m_{ij}\mathbf{P}_i$, where $j \in [1, \lceil \frac{\lambda}{s} \rceil]$. The relation between the original task and these subtasks can be expressed by Formula (1).

$$\begin{aligned} \mathbf{Q} = \sum_{i=1}^{n} k_i\mathbf{P}_i &= \sum_{i=1}^{n} \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil} (2^{(j-1)s} m_{ij})\mathbf{P}_i \\ &= \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(j-1)s} \left(\sum_{i=1}^{n} m_{ij}\mathbf{P}_i \right) \\ &= \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(j-1)s} \mathbf{G}_j \end{aligned} \tag{1}$$

2) Compute subtask results $\mathbf{G}_j$, where $j \in [1, \lceil \frac{\lambda}{s} \rceil]$. For each subtask, as shown in Figure 4, it puts EC points $\mathbf{P}_i$ with the same scalar value m_{ij} into a bucket whose index is equal to m_{ij}. Note that only $2^s - 1$ buckets need to be prepared because the points corresponding to zero scalars have no effect on the final result and are skipped directly. Then, it adds up (PADD) all points in the same buckets. The sum point of each bucket is called the bucket point, denoted as $\mathbf{B}_l$, where l is the bucket index and $l \in [1, 2^s - 1]$.

The subtask result is exactly equal to the sum of all bucket points weighted by their bucket indexes, namely $\mathbf{G}_j = \sum_{l=1}^{2^s-1} l\mathbf{B}_l$. Next, it uses an efficient approach proposed in [BDLO12] to compute $\sum_{l=1}^{2^s-1} l\mathbf{B}_l$, as shown in Algorithm 2. In short, it starts by calculating a serial of new points $\mathbf{M}_l = \sum_{u=2^s-l}^{2^s-1} \mathbf{B}_u$ with a recursive method given by the Formula $\mathbf{M}_l = \mathbf{M}_{l-1} + \mathbf{B}_{2^s-l}$, where $l \in [1, 2^s - 1]$ and the start point $\mathbf{M}_1 = \mathbf{B}_{2^s-1}$. The subtask result $\mathbf{G}_j$ can be obtained by adding up all new points $\mathbf{M}_l$ via Formula (2).

$$\sum_{l=1}^{2^s-1} \mathbf{M}_l = \sum_{l=1}^{2^s-1} \sum_{u=2^s-l}^{2^s-1} \mathbf{B}_u = \sum_{l=1}^{2^s-1} l\mathbf{B}_l = \mathbf{G}_j \tag{2}$$

3) Compute the MSM result with the subtask results, namely $\mathbf{Q} = \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(j-1)s}\mathbf{G}_j$. Specifically, it calculates a serial of new points $\mathbf{T}_u = \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil - u + 1} 2^{(j-1)s}\mathbf{G}_{j+u-1}$ with an inverse recursive method via Formula (3), where $u \in [1, \lceil \frac{\lambda}{s} \rceil]$ and the end point $\mathbf{T}_{\lceil \frac{\lambda}{s} \rceil} = \mathbf{G}_{\lceil \frac{\lambda}{s} \rceil}$. Finally, the MSM result $\mathbf{Q}$ is exactly equal to $\mathbf{T}_1$. The computational costs of this recursive method are lower than that of using Formula (1) directly.

$$\mathbf{T}_u = 2^s\mathbf{T}_{u+1} + \mathbf{G}_u \tag{3}$$

Complexity. For each subtask, it requires at most n PADDs to put all points into the buckets and $(2^{s+1} - 2)$ PADDs to get the subtask result using Algorithm 2. In order to add the subtask results to the final result, a recursive method based on Formula (3) is used, which requires around s PDBLs and 1 PADD per subtask on average. Since there are $\lceil \frac{\lambda}{s} \rceil$ subtasks, the total computational costs of the Pippenger algorithm are around $\lceil \frac{\lambda}{s} \rceil (n + 2^{s+1})$ PADDs plus λ PDBLs. Note that we skip the costs of scalar operations here because they are negligible compared to the costs of EC point operations.

2.4 Sparse Matrix

Sparse matrices have a significant impact on our work in improving the efficiency of zkSNARK. Here, we present their storage formats and the basic operations they support.

CSR format and ELL format are two of the most popular sparse matrix storage formats. Examples of these two formats are shown in Figure 5. The ELL format consists of three structures, `Data`, `ColIndex`, and `RowLength`. Specifically, the nonzero elements in the same row of the sparse matrix are stored in the same row of the `Data`. The `ColIndex` stores the column indices of these nonzero elements. All rows of the `Data` and `ColIndex` structures are padded to length `RowSpace` to meet the alignment requirement. The `RowLength` stores the number of the nonzero elements in each row of the sparse matrix. The CSR format also consists of three structures, `Data`, `ColIndex`, and `RowPtr`. The first two structures are the same as the two in the ELL format, except that they do not need to meet the alignment requirements. The `RowPtr` is an array of length `RowNum` + 1. Its i-th element encodes the cumulative number of nonzero elements up to the i-th row, where $i \in [0, \texttt{RowNum}]$.

The basic operations supported by sparse matrices include sparse matrix transposition, sparse matrix-vector multiplication, and so on. Many well-studied GPU implementations [BG09, GD14, TDM+14] are available for speeding up sparse matrix basic operations, where they achieve high performance based on classical GPU optimization methods, including loop unrolling, load balancing, and coalescing memory accesses. Moreover, these GPU implementations have been deployed in many industrial libraries [NCVK10, DBOG14]. Therefore, converting other complex operations to basic sparse matrix operations is commonly a suitable and convenient choice to improve the efficiency of the computation.

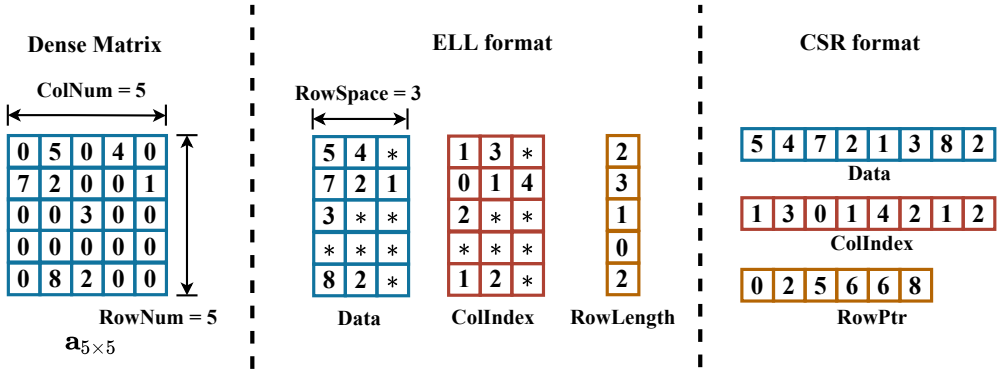

Figure 5: Sparse matrix representations for a simple example matrix $\mathbf{a}_{5\times5}$.

2.5 Graphics Processing Units

Graphics Processing Units (GPUs) are many-core computing platforms that support the concurrent execution of multiple threads. A typical GPU consists of multiple Streaming Multiprocessors (SMs) and a global memory. Each SM includes multiple Scalar Processors (SPs), a shared memory, and several on-chip registers. These registers and various kinds of memory constitute the multiple memory hierarchy architecture of GPUs. The on-chip registers are the fastest memory component but have minimal storage capacity, while the global memory provides the largest storage capacity but is the slowest. The performance of the shared memory is between the on-chip registers and the global memory.

Another special thing about GPUs is their execution fashion. Warps instead of threads are the basic execution units on GPUs. Each warp consists of 32 threads and is scheduled by warp schedulers residing in SMs. Specifically, each warp scheduler maintains a list of active warps and picks a warp from the list on each cycle to execute an instruction. Threads in a warp can carry their own private data but have to execute the same instructions. This execution fashion is known as the Single Instruction Multiple Thread (SIMT).

3 A Faster Parallel MSM Algorithm

Multi-scalar multiplication (MSM) is defined as $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$, where k_i is a λ-bits scalar, $\mathbf{P}_i$ is an EC point, and the pair $k_i\mathbf{P}_i$ represents point scalar multiplication of k_i and $\mathbf{P}_i$. Due to MSM being the most time-consuming operation in zkSNARK, an efficient parallel MSM algorithm can greatly improve its efficiency. In this section, we introduce some naive parallel MSM approaches and present our proposed parallel MSM algorithm.

3.1 Some Naive Approaches

Thanks to the outstanding performance of the Pippenger algorithm, parallel MSM methods based on it have been experimentally shown to perform better than other MSM algorithms. Therefore, efficient industrial implementations [Lib14, Gna20, WZC+18, Bel15, Bel19] of MSM are now choosing to be based on this algorithm. Here, we briefly introduce three naive parallel Pippenger-based approaches and then give their computational costs.

Recall that in the Pippenger algorithm, it divides the original MSM task into $\lceil \frac{\lambda}{s} \rceil$ subtasks. The first naive approach leverages this feature. It parallelizes the MSM computation by the observation that all subtasks in the serial Pippenger algorithm can be performed simultaneously. Therefore, it arranges $\lceil \frac{\lambda}{s} \rceil$ threads to perform these subtasks simultaneously. After all threads obtain the subtask results, one of the threads adds these results to the final result based on Formula (3). However, this parallel algorithm at most

provides a speedup of $\lceil \frac{\lambda}{s} \rceil$, which is much less than the number of cores GPU provides. Therefore, it is not suitable for the GPU implementation of MSM. Note that λ typically ranges from 254 to 768, and s can be chosen at will.

The second naive approach is a more general parallel method. It decomposes the original MSM computation into t parts, where t is the total number of threads. Each part is a small-scale MSM computation, namely $\mathbf{Q}_j = \sum_{i=1}^{\hat{n}} k_{j\hat{n}+i}\mathbf{P}_{j\hat{n}+i}$, where $j \in [0, t-1]$ and $\hat{n} = \frac{n}{t}$. Next, all threads perform the serial Pippenger algorithm for their corresponding small-scale MSM computation. The final result $\mathbf{Q} = \sum_{j=1}^{t} \mathbf{Q}_j$ can be obtained with the parallel sum algorithm. Recall that the advantage of the Pippenger algorithm is to compute large-scale MSMs. However, here it decomposes the large-scale MSM into multiple small-scale MSMs, which obviously weakens the advantage of the Pippenger algorithm.

The third naive approach combines the above two parallel algorithms. First, this algorithm decomposes the original MSM computation into $t/\lceil \frac{\lambda}{s} \rceil$ small-scale MSM computations similar to the second parallel algorithm. Next, for each small-scale computation, it schedules $\lceil \frac{\lambda}{s} \rceil$ threads to perform the first parallel algorithm. The final result can be obtained by adding up all results of the $t/\lceil \frac{\lambda}{s} \rceil$ small-scale computations. The performance of this algorithm is better than the above two algorithms in the case of high parallelism, but it still cannot achieve perfect linear speedup over the serial Pippenger algorithm, where perfect linear speedup means the parallel speedup ratio is equal to the number of execution threads.

Complexity. The computational costs of the first algorithm are around $n + 2^{s+1} + \lceil \frac{\lambda}{s} \rceil$ PADDs plus λ PDBLs for each thread when the number of threads t is larger than $\lceil \frac{\lambda}{s} \rceil$; the computational costs of the second algorithm are around $\lceil \frac{\lambda}{s} \rceil (\frac{n}{t} + 2^{s+1}) + \log t$ PADDs plus λ PDBLs for each thread; the computational costs of the third algorithm are around $\lceil \frac{\lambda}{s} \rceil (\frac{n}{t}) + 2^{s+1} + \lceil \frac{\lambda}{s} \rceil + \log(t/\lceil \frac{\lambda}{s} \rceil)$ PADDs plus λ PDBLs for each thread. Note that we also skip the costs of scalar operations here because they are negligible compared to the costs of EC point operations.

3.2 Our Parallel MSM Algorithm

Our proposed parallel MSM algorithm is also inspired by the Pippenger algorithm. However, we do not use the parallel methods that decompose the large MSM computation into multiple smaller ones like the second and the third naive approaches, because decomposing the large MSM computation can weaken the advantage of the Pippenger algorithm. More specifically, we notice the advantage of the Pippenger algorithm only comes when there are a great amount of EC points placed into the same buckets and processed as a whole. The larger the scale of MSMs is, the more benefits this advantage brings. Thus, decomposing the large MSM computation is obviously unsuitable for the Pippenger-based MSM algorithm, which manifests an unsatisfiable increase in speedup with the increasing parallelism; see the comparison between the computational costs given in Section 2.3 and Section 3.1. So we need to find a parallel algorithm that processes the MSM computation from a global perspective.

Sparse matrices are excellent tools for integrating fragmented elements into a whole without consuming too much storage space. Hence, we propose a new parallel MSM algorithm with the help of sparse matrices to strengthen the advantage of the Pippenger algorithm. Our algorithm is well-suitable for execution in GPUs and has nearly perfect linear speedup over the Pippenger algorithm. Below we give an overview of our algorithm and then present its details.

Overview. Given λ-bits scalars $k_1, ..., k_n$ and base points $\mathbf{P}_1, ..., \mathbf{P}_n$, we choose a window size s and convert the original MSM task to $\lceil \frac{\lambda}{s} \rceil$ subtasks like the Pippenger algorithm. Then, we compute each subtask $\sum_{i=1}^{n} m_i \mathbf{P}_i$. Specifically, as shown in Figure 6, we first divide EC points into t parts. For each part, we add and store the points with the

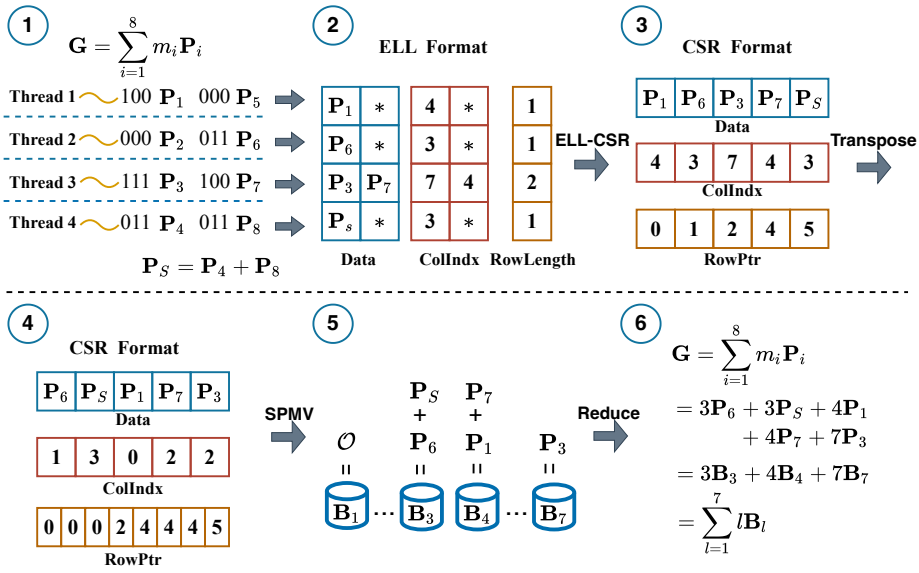

Figure 6: An simple example of our parallel MSM algorithm.

same scalar value into the same entry of a sparse matrix. This matrix is in ELL format with t rows and $2^s - 1$ columns, where t is the total number of threads. After the above step is completed, we convert this sparse matrix from ELL format to CSR format and then transpose it. Next, we perform the sparse matrix-vector multiplication (SPMV) on the transposed matrix with a scalar vector whose elements are all equal to 1. The result is the EC point vector consisting of points $\mathbf{B}_1, ..., \mathbf{B}_{2^s-1}$. Finally, we compute $\sum_{l=1}^{2^s-1} l\mathbf{B}_l$, which is equal to the result of the subtask $\sum_{i=1}^{n} m_i \mathbf{P}_i$. Note that all the above steps can be parallelized with limited load imbalance. Details are described below and shown in Algorithm 3.

Details. Based on the Pippenger algorithm, we start by converting the original task $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$ into $\lceil \frac{\lambda}{s} \rceil$ subtasks $\mathbf{G}_j$, where s is the chosen window size as in the Pippenger algorithm and $j \in [1, \lceil \frac{\lambda}{s} \rceil]$. The relation between the original task and subtasks can be expressed by Formula (1). Next, we execute these subtasks serially. For each subtask, we do the following two steps:

1) Store all EC points $\mathbf{P}_i$ into a sparse matrix. We begin with generating an empty sparse matrix with t rows and $2^s - 1$ columns, where t is the total number of threads. This sparse matrix is in ELL storage format and its `RowSpace` is $\frac{n}{t}$. Here, we use the ELL storage format because it is efficient to store EC points in this format matrix parallelly. Specifically, as shown in the first step of Figure 6, we divide EC points into t parts. For each part, EC points with the same scalar value are added and stored in the same entry of a row by a thread. The column index of this entry is set to the scalar value. Note that the points corresponding to zero scalars have no effect on the final result and can be skipped.

2) Get an EC point vector, whose elements play a similar role as bucket points in the Pippenger algorithm. This EC point vector is donated as $\overrightarrow{\mathbf{B}}^{\langle j \rangle}_{2^s-1}$, where j is the sequence number of the subtask. Firstly, we convert the sparse matrix from ELL format to CSR format and transpose it in parallel. The reason that we employ CSR format is to save space costs, since the alignment requirement of ELL format leads to additional space overhead for storing the matrix. Next, we add up all EC points that are in the same row of the transposed matrix. This operation is equivalent to performing the sparse matrix-vector multiplication on the matrix with a scalar vector whose elements are all equal to 1. The

Algorithm 3 Our Parallel MSM Algorithm

Require: A scalar vector $\overrightarrow{k}_n = [k_1, k_2, ..., k_n]$, whose elements are λ-bit scalars. A point vector $\overrightarrow{\mathbf{P}}_n = [\mathbf{P}_1, \mathbf{P}_2, ..., \mathbf{P}_n]$. A chosen window size s. The number of threads t. $\lceil \frac{\lambda}{s} \rceil$ GPU streams $[stream_1, stream_2, ..., stream_{\lceil \frac{\lambda}{s} \rceil}]$.

Ensure: $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$

for $j \leftarrow 1$ to $\lceil \frac{\lambda}{s} \rceil$ **do** *// Convert the original task into $\lceil \frac{\lambda}{s} \rceil$ subtasks.*
  $row\_num \leftarrow t$; $col\_num \leftarrow 2^s - 1$; $row\_space \leftarrow \frac{n}{t}$
  $ell \leftarrow$ GenELLMtx$(row_num, col_num, row_space)$
  // m_i is a part of k_i used in this subtask
  for $i \leftarrow 1$ to n **do** in parallel with t threads in $stream_1$
    $m_i \leftarrow (k_i \gg ((j-1) * s)) \ \& \ ((1 \ll s) - 1)$
  end for
  SynchronizeThreadsInStream$(stream_1)$ *// use the* cudaStreamSynchronize *function.*
  $\overrightarrow{m}_n \leftarrow [m_1, m_2, ..., m_n]$
  // Store EC points into the sparse matrix, as shown in the first step of Figure 6.
  $ell \leftarrow$ pStoreECPoints$(ell, \overrightarrow{m}_n, \overrightarrow{\mathbf{P}}_n, t, stream_1)$
  $csr \leftarrow$ pELL2CSR$(ell, t, stream_1)$
  $csr \leftarrow$ pTranspose$(csr, t, stream_1)$
  $\overrightarrow{v}_t \leftarrow [1, 1, ..., 1]_t$ *// A scalar vector whose elements are all equal to 1.*
  $\overrightarrow{\mathbf{B}}^{\langle j \rangle}_{2^s-1} \leftarrow$ pSparseMatrixVectorMUL$(csr, \overrightarrow{v}_t, t, stream_1)$
end for
for $j \leftarrow 1$ to $\lceil \frac{\lambda}{s} \rceil$ **do** in parallel
  $\mathbf{G}_j \leftarrow$ pBucketPointsReduction$(\overrightarrow{\mathbf{B}}^{\langle j \rangle}_{2^s-1}, t/\lceil \frac{\lambda}{s} \rceil, stream_j)$ *// See Algorithm 4.*
end for
// The following loop is to synchronize all t threads launched in the above loop.
for $j \leftarrow 1$ to $\lceil \frac{\lambda}{s} \rceil$ **do**
  SynchronizeThreadsInStream$(stream_j)$ *// use the* cudaStreamSynchronize function.
end for
$\mathbf{T}_{\lceil \frac{\lambda}{s} \rceil} \leftarrow \mathbf{G}_{\lceil \frac{\lambda}{s} \rceil}$
for $j \leftarrow (\lceil \frac{\lambda}{s} \rceil - 1)$ to 1 **do**
  $\mathbf{T}_j \leftarrow 2^s \mathbf{T}_{j+1} + \mathbf{G}_j$ *// Add $\mathbf{G}_j$ to the final result with Formula (3).*
end for
$\mathbf{Q} \leftarrow \mathbf{T}_1$
return Q

result is the EC point vector that we need. Note that the above parallel sparse matrix operations are well-studied [BG09, GD14, TDM+14].

After obtaining the EC point vectors of all subtasks, we schedule $t/\lceil \frac{\lambda}{s} \rceil$ threads for each subtask to compute the sum of all points $\mathbf{B}_l^{\langle j \rangle}$ weighted by their indexes l with Algorithm 4, whose results are exactly equal to the subtask results, namely $\mathbf{G}_j = \sum_{l=1}^{2^s-1} l\mathbf{B}_l^{\langle j \rangle}$. Finally, we can get the final result $\mathbf{Q}$ by adding all subtask results based on Formula (3).

To guarantee correct calculations in Algorithm 3 and Algorithm 4, we use stream barriers, implemented by the `cudaStreamSynchronize` function, to synchronize all launched threads. These barriers in our algorithm provide the same functionality as global barriers while maintaining compatibility with the multi-stream technology used in Section 4.4. Although these barriers force the program to wait until all threads finish their tasks, the overhead is limited due to the nearly even distribution of workload across threads. In addition, n and $\lceil \frac{\lambda}{s} \rceil$ are not required to be multiples of the warp size, thanks to these stream barriers.

Algorithm 4 pBucketPointsReduction

Require: EC point vectors $\overrightarrow{\mathbf{B}}_{2^s-1} = [\mathbf{B}_1, \mathbf{B}_2, ..., \mathbf{B}_{2^s-1}]$. The number of threads t. A GPU stream *stream*.
Ensure: An EC point $\mathbf{G} = \sum_{l=1}^{2^s-1} l\mathbf{B}_l$.

$\xi \leftarrow$ GetThreadID() // *Thread ID,* $\xi \in [1, t]$.
// *Divide* $2^s - 1$ *vector elements into* t *parts. Each part has* r *EC points.*
$r \leftarrow (2^s - 1)/t$
for $l \leftarrow 1$ to r **do**
  if $l \neq 1$ **then**
    $\mathbf{M}_{(\xi-1)r+l} \leftarrow \mathbf{M}_{(\xi-1)r+l-1} + \mathbf{B}_{\xi r+1-l}$
    $\mathbf{S}_\xi \leftarrow \mathbf{S}_\xi + \mathbf{M}_{(\xi-1)r+l}$
  else
    $\mathbf{M}_{(\xi-1)r+l} \leftarrow \mathbf{B}_{\xi r}$
    $\mathbf{S}_\xi \leftarrow \mathbf{M}_{(\xi-1)r+l}$
  end if
end for
// *After completing the above loop,*
// $\mathbf{S}_\xi = \mathbf{B}_{(\xi-1)r+1} + 2\mathbf{B}_{(\xi-1)r+2} + ... + r\mathbf{B}_{(\xi-1)r+r}$
// $\mathbf{M}_{\xi r} = \mathbf{B}_{(\xi-1)r+1} + \mathbf{B}_{(\xi-1)r+2} + ... + \mathbf{B}_{(\xi-1)r+r}$
$\mathbf{S}_\xi \leftarrow \mathbf{S}_\xi + ((\xi - 1)r)\mathbf{M}_{\xi r}$
SynchronizeThreadsInStream(*stream*) // *use the* cudaStreamSynchronize *function.*
$\overrightarrow{\mathbf{S}}_t \leftarrow [\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_t]$
$\mathbf{G} \leftarrow$ pSum($\overrightarrow{\mathbf{S}}_t, t$, *stream*) // $\mathbf{G} = \mathbf{S}_1 + \mathbf{S}_2 + ... + \mathbf{S}_t$
return G

Complexity. The computational costs of storing EC points into the sparse matrix and the computational costs of the sparse matrix-vector multiplication vary with the scalar vector $\overrightarrow{k}_n$. However, the total computational costs of these two parts are fixed. They are at most $\lceil \frac{\lambda}{s} \rceil$ n PADDs in total, and thus each thread needs to perform $\lceil \frac{\lambda}{s} \rceil (\frac{n}{t})$ PADDs on average. PMULT is not needed because all elements of the vector used in the matrix-vector multiplication are equal to 1. Note that the computational load on each thread may be imbalanced here, because a naive sparse matrix-vector multiplication (SPMV) method cannot guarantee the workload of each thread is the same. Fortunately, this problem can be mitigated with our proposed SPMV approach, as shown in Section 4.1. After obtaining the EC point vectors of all subtasks, it requires at most $\lceil \frac{\lambda}{s} \rceil (\frac{2^{s+1}}{t} - 1) + s + \log t$ PADDs and s PDBLs for each thread to get subtask results with Algorithm 4. Finally, in order to add subtask results to the final result, a recursive method implied by Formula (3) is used, which takes less than $\lceil \frac{\lambda}{s} \rceil$ PADDs and λ PDBLs. Therefore, the total computational costs for each thread are around $\lceil \frac{\lambda}{s} \rceil (\frac{n}{t} + \frac{2^{s+1}}{t}) + s + \log t$ PADDs plus $\lambda + s$ PDBLs. The values of s and $\log t$ are both small. Therefore, our MSM algorithm has nearly perfect linear speedup over the Pippenger algorithm, whose computational costs are around $\lceil \frac{\lambda}{s} \rceil (n + 2^{s+1})$ PADDs plus λ PDBLs. Here we skip the computational costs of ELL-CSR format conversion and sparse matrix transpose because they only require some scalar operations and data movement operations, whose costs are negligible compared to the costs of EC point operations.

4 An Efficient GPU Implementation of zkSNARK

This section presents cuZK, an efficient GPU implementation of zkSNARK targeting the selected Groth's protocol. Here, we also need to mention that our techniques are adapted to similar zkSNARK protocols, especially for the protocols [GGPR13, GM17, GWC19,

CHM+20] that require the multi-scalar multiplication operation. Below, we present the GPU implementations of three time-consuming operations in Groth's protocol, namely multi-scalar multiplication in Section 4.1, matrix-vector multiplication in Section 4.2, and number-theoretic transform in Section 4.3. Then, the overall dataflow of cuZK is given in Section 4.4.

4.1 Multi-scalar Multiplication on GPUs

Here, we implement the parallel MSM algorithm proposed in Section 3.2 and provide further optimizations to make it suitable for running on GPUs. Below, we present crucial parts of our GPU implementation.

Choose an optimal window size: Recall that our parallel MSM algorithm requires choosing a window size s. The relationship between the computational costs of our MSM algorithm and the window size s is given in the complexity part of Section 3.2 by the formula $\lceil \frac{\lambda}{s} \rceil (\frac{n}{t} + \frac{2^{s+1}}{t}) + s + \log t$ PADDs plus $\lambda + s$ PDBLs, where n is the scale of MSM, λ is the number of scalar bits, t is the number of parallel threads, and s is the window size. To simplify the analysis, we assume that the computational costs of PADD and PDBL are equivalent. By making this assumption, we can treat the above formula as a mathematical function, where the window size s serves as the independent variable. Therefore, after fixing the MSM parameters and the number of parallel threads (equal to the GPU core count), we determine the optimal window size s by traversing all feasible window sizes and finding the one for which our MSM algorithm has the minimum computational costs. For example, when computing MSM with the number of scalar bits $\lambda = 255$ and the scale $n = 2^{20}$ in a V100 card with $5,120$ GPU cores, we can explore window sizes from 1 to 255 and find the value $s = 16$ that makes the above function of the computational costs having the minimum value. Note that this step of finding the optimal window size can be performed offline without affecting the performance of the MSM computation.

Put EC points into a sparse matrix: First, we allocate space for a sparse matrix of ELL format on the global memory so that every thread can access this matrix. Then, each thread should have stored EC points into the sparse matrix as in Figure 6. However, in practice, what each thread store in the sparse matrix is not the EC points themselves but their indexes in the EC point vector. Roughly, each EC point typically has hundreds of bits, while the index size is the logarithm of the vector scale, only tens of bits. Thus, this step significantly saves the device storage costs.

Format conversion and transposition for the sparse matrix: As mentioned in Algorithm 3, the next step we need to perform is converting the sparse matrix to the CSR format and then transposing it in parallel. Format conversion is straightforward. We only need to remove the empty positions of `Data` and `ColIndex` for row alignment. `RowPtr` in the CSR format is actually the prefix sum of `RowLengh` in the ELL format.

Next, for the sparse matrix transposition, there are various existing efficient GPU implementations. For example, a scheme based on radix sort performs well. For convenience, we refer to this scheme as the sort-based scheme. Actually, this sort-based scheme also appears in two MSM implementations [Yrr22, Mat22], which are both concurrent works with this paper. The main difference is that they do not abstract the sort-based scheme into a sparse matrix transpose or decouple it from the MSM algorithm, while our work views this sort-based scheme as a concrete scheme for the sparse matrix transpose, that is, we can easily replace it with a faster sparse matrix transpose scheme in the future.

As we currently implement this sorting-based scheme as well, we give its details below. First, we get row positions of elements in the sparse matrix using `RowPtr` and store those positions in a new structure `RowIndex`. Then we sort the triplet <`ColIndex`, `Data`, `RowIndex`> with `ColIndex` as the sorting key. The sorted `Data` is the `Data` of the transpose matrix and the sorted `RowIndex` is the `ColIndex` of the transpose matrix. The `RowPtr` of

the transpose matrix can be obtained by performing run-length encoding and prefix sum operation to the sorted `ColIndex`.

Fetch EC points from host memory: Afterward, we fetch the corresponding EC points from host memory to device memory according to the indices stored in the matrix. A naive method of moving EC points from host memory to device memory takes much time on data transfer. Fortunately, its latency overhead can be almost eliminated by overlapping CPU-GPU data transfer and device computing based on the multi-streaming technique. The details of the overlap are described in Section 4.4.

Perform sparse matrix-vector multiplication: We sum up the EC points that are in the same row of the transposed matrix. The summation step is actually equivalent to performing parallel sparse matrix-vector multiplication (SPMV) on the matrix with a scalar vector whose elements are all equal to 1. This step may introduce severe thread load imbalance due to the different lengths of the matrix rows. To overcome load imbalance, we propose a GPU-based SPMV implementation called CSR-Balanced.

Specifically, CSR-Balanced overcomes load imbalance by dynamically scheduling different numbers of threads to work on different matrix rows. It first sorts the matrix rows and divides them into different groups based on the row lengths. Then, it only allows GPU warps instead of individual threads to work across these groups, and thus all threads in a GPU wrap have to work in the same group. This step guarantees that the workload of all threads in a warp is almost balanced because rows in the same group have similar lengths. Next, in order to balance the workload of each warp, CSR-Balanced schedules different numbers of wraps for groups according to the proportion of non-zero matrix entries in each group so that the number of non-zero entries that each warp works on is similar. The additional overhead of this method is the sorting costs, which are negligible compared to the costs of EC point operations. Note that CSR-Balanced cannot be used in SPMV for regular scalar operations because the sorting costs are relatively high compared to the costs of regular scalar operations.

Bucket point reduction: In this part, we follow the description in Algorithm 4. The only crucial part we need to be mentioned is the parallel sum operation. It is a basic reduction operation commonly used in parallel programming to add up all vector elements and is deployed in various GPU libraries. Speficically, the parallel summation operation adopts the tree reduction method. First, $t/2$ threads are required to add (PADD) t EC points (two elements per thread) and then recursively halve the number of threads to add the previous step's results until a single aggregate is obtained. This single aggregate is the sum of t EC points.

Multi-GPU implementation: The efficiency of MSM can be further accelerated with multiple GPUs. We give the multi-GPU implementation of our MSM algorithm. Recall that our parallel MSM algorithm decomposes the original MSM task into multiple subtasks. Therefore, for multi-GPU implementation, we assign these subtasks to GPUs evenly. GPUs use the same implementation following the operations described above to calculate the subtask results. Once these subtask results are obtained, all GPUs transfer them to a single GPU. Communication between GPUs can be performed either by using the CPU memory as an intermediate transfer station or through Nvidia NVLink for direct GPU-GPU transfer. Nvidia NVLink is a wire-based interconnect technology, and it enables direct GPU-GPU data transfer by employing the unified memory or using memory management functions such as `cudaMemcpyPeer` and `cudaMemcpy` with the `cudaMemcpyDeviceToDevice` flag. All these methods only work after the `cudaDeviceEnablePeerAccess` function is performed. Finally, the GPU that receives all subtask results adds them up to get the final result using Formula (3). Alternatively, since the last step involving Formula (3) is executed serially, we can also opt to transfer all subtask results obtained on GPUs to the CPU memory through functions like `cudaMemcpy` with `cudaMemcpyDeviceToHost` flag. Then, the CPU adds these results to get the final result. Note that each subtask result is an

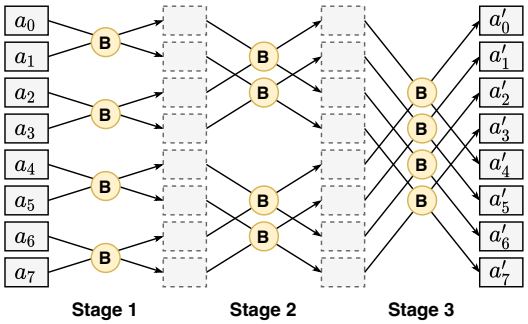

Figure 7: The butterfly diagram for an 8-point NTT. B represents the butterfly operation.

EC point, which only has hundreds of bits. Therefore, our multi-GPU implementation does not introduce substantial additional transfer overhead compared to our single-GPU implementation.

4.2 Matrix-vector Multiplication on GPUs

In Groth's protocol, matrix-vector multiplication (MUL) operates on the R1CS matrix that is compiled from the function to be proved, as shown in Section 2.1. The computational costs of MUL mainly depend on the scale of the matrix it operates on. In real-world applications, the R1CS matrix is commonly very large but sparse. For example, in Filecoin [BG17], only less than 0.1% the matrix entries are non-zero. Therefore, we choose the CSR storage format to store the R1CS matrix. This step helps to reduce the storage costs and move the whole R1CS matrix to the GPU memory.

After the R1CS matrix is moved into the GPU memory, we perform the MUL computation with the parallel schemes for the sparse matrix. There are many parallel sparse matrix-vector multiplication (SPMV) schemes, including CSR-Scalar [Gar08], CSR-Vector [BG09], and CSR-Balanced proposed in Section 4.1. However, none of these schemes can be suitable for sparse matrices with different characteristics. For example, CSR-Scalar arranges each thread to work on each row of the sparse matrix. This scheme may cause severe load imbalance when the variance of matrix row lengths is very large. Therefore, we cannot choose only one static parallel scheme for the MUL computation.

In our MUL implementation, we employ different SPMV schemes for different R1CS matrices. For a specific R1CS matrix, we first count out characteristics of the R1CS matrix, such as the variance and mean of its row lengths. Then, we choose CSR-Scalar for the R1CS matrix with small variance and small mean, CSR-Vector for the R1CS matrix with small variance and large mean, and CSR-Balanced for the R1CS matrix with large variance. The above method can avoid the drawbacks of these parallel SPMV schemes. In addition, the operation for the sort operation existing in CSR-Balanced and the matrix characteristics calculation can actually be performed offline because the R1CS matrix that MUL operates on is infinitely reusable for the function to be proved. Therefore, this method does not introduce additional overhead for the MUL computation online.

4.3 Number-theoretic Transform on GPUs

The number-theoretic transform (NTT) is essentially discrete fourier transform (DFT) over finite fields. It is defined as the transform between two N-sized vectors $\overrightarrow{a}'_N \stackrel{\text{def}}{=} \mathsf{NTT}(\overrightarrow{a}_N)$ with their elements $a'_i = \sum_{j=0}^{N-1} a_j \omega_N^{ij}$, where a'_i and a_j are λ-bits scalars in a finite field and ω_N is the Nth root of unity in the same field. The exponents of ω_N are called *twiddle factors*. The inverse number-theoretic transform (INTT) is the inverse transformation of

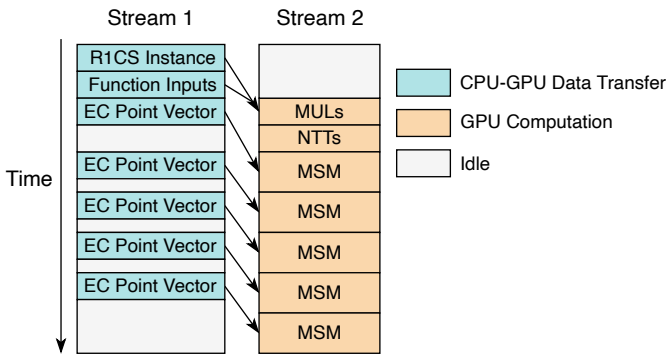

Figure 8: Timeline for the execution of cuZK.

NTT. It can be easily completed by NTT with different twiddle factors. Actually, NTT is a critical module commonly used in cryptography. Therefore, there are many efficient GPU implementations [KJPA20, GXW21] that have been developed for its computation. For instance, a state-of-the-art implementation can be found in [GJCC20], which was originally used in post-quantum encryption algorithms. Actually, we can easily retrofit this NTT implementation so that it is adapted to the setting of zkSNARK.

For more details, similar to the standard DFT algorithms [CT65], we decompose the overall computation of NTT into $\log N$ stages, where each stage requires $N/2$ *butterfly* operations [Opp99]. A single butterfly operation performs reading two input values, processing input values, and storing results. Figure 7 shows an example of the butterfly diagram for an 8-point NTT. We can see that the butterfly operations at each stage are independent. Therefore, we can parallelize NTT by launching $N/2$ threads to perform these butterfly operations concurrently. Note that we hold the results at the intermediate stages in the global memory of GPUs because faster registers and shared memory are not large enough to accommodate these intermediate results in zkSNAKR. The final results of NTT are exactly the results at the last stage.

4.4 Overall Dataflow of cuZK

With our self-designed MSM and MUL parallel schemes and the well-studied NTT parallel scheme, we have achieved the parallelization of all zkSNARK operations. Moreover, these parallel schemes are well-suitable for execution on GPUs. Therefore, to make the best use of these parallel schemes, we perform all operations of zkSNARK on GPUs, which additionally eliminates redundant CPU-GPU data transfer. Note that these operations also include the operations with small computational costs, like variable initialization. We can perform them with small kernels that only launch one thread. These small kernels have very little impact on the overall performance due to their small computational costs. As a result, only three storage modules need to be sent to GPUs, namely the R1CS instance, the function inputs, and the prover key. We overlap data transfer and device computation using the multi-streaming technique to further reduce the latency overhead caused by CPU-GPU data transfer. Dataflow is shown below.

We first transmit the R1CS instance and the function inputs from CPU to GPU. As stated in Section 2.1, the R1CS instance consists of three matrices in CSR format, and the function inputs make up a vector whose elements include all intermediate results of the compiled function. Due to the above two storage modules being required by the MUL operation, the first operation performed in the proof generation step, we have to finish this transfer before the device computation begins. Another essential storage module, the prover key, consists of multiple large-scale EC point vectors and thus is very large

Table 1: Hardware Configuration of Testbeds.

Testbeds	V100	G3060	VU9P
Device	Tesla V100	GTX 3060	UltraScale+ VU9P
Platform	GPU	GPU	FPGA
Core Count	8 × 5120	3584	/
Clock	1.24 GHz	1.32 GHz	0.28 GHz
Host (CPU)	Xeon(R) Platinum 8260	Ryzen 3700X	Xeon(R) E5-2686
CPU Cores	2 × 24	8	8 (vCPU)
CPU Freq.	2.40 GHz	3.60 GHz	2.30 GHz
OS	CentOS 7.8	Ubuntu 20.04	Amazon Linux 2

Table 2: Some Baseline Implementations.

Implementations	Platform	Multi-PUs [(1)]	Supported Operations [(2)]	Optional Elliptic Curves
cuZK (ours)	GPU	✓	Groth	BLS12-381, MNT4753, BLS12-377
Bellperson [Bel19]	GPU	✓	Groth	BLS12-381
Mina [Min19]	GPU	×	Groth	MNT4753
Yrrid [Yrr22]	GPU	×	MSM	BLS12-377
MatterLab [Mat22]	GPU	×	MSM	BLS12-377
Bellman [Bel15]	CPU	×	Groth	BLS12-381
Hardcaml [Har22]	FPGA	×	MSM, NTT	BLS12-377

[(1)] This label represents whether these implementations support multi-CPU/GPU execution or not.
[(2)] This label represents the operations supported by these implementations, where Groth represents all operations in Groth's protocol.

in size. Moving the prover key to GPUs takes a lot of time and also occupies a large amount of GPU memory resources. Therefore, we choose to overlap its transfer and device computation with the multi-streaming technique. As shown in Figure 8, we overlap the MULs and NTTs computation with the first MSM-required EC points transfer, the first MSM computation with the second MSM-required EC points transfer, the second MSM computation with the third MSM-required EC points transfer, up to the second-to-last MSM computation with the last MSM-required EC points transfer. Moreover, in order to save storage costs and adapt to large-scale MSM, cuZK frees the corresponding memory when the whole EC point vector or its some elements is no longer used. This overlapping approach eliminates almost all latency overhead caused by the data transfer of the prover key. Finally, the proof can be obtained by simply processing the results of MSM.

5 Evaluation

In this section, we first give our experimental setting in Section 5.1. Then, the evaluation results are presented. Specifically, we give the benchmark results for our MSM implementation in Section 5.2. This aims to show the improvement that our parallel MSM algorithm provides exclusively. Next, the overall performance of cuZK is shown in Section 5.3.

5.1 Experimental Setup

We perform the experiments on three testbeds: 1) V100, 2) G3060, and 3) VU9P, whose hardware configurations are shown in Table 1. The testbed V100 is equipped with an

Intel(R) Xeon(R) Platinum 8260 CPU chip and eight Nvidia Tesla V100 GPU cards. All GPU cards are connected with Nvidia NVLink, which is a near-range efficient interconnect supported by physical wires. After the `cudaDeviceEnablePeerAccess` function is performed, NVLink facilitates direct GPU-GPU data transfer by employing the unified memory or using memory management functions such as `cudaMemcpyPeer` and `cudaMemcpy` with the `cudaMemcpyDeviceToDevice` flag. This feature can be used for the method that directly transfers the subtask results from multiple GPUs to a single GPU in our multi-GPU implementation part of Section 4.1. Our experiments on multi-GPU systems are executed on the testbed V100. The testbed G3060 is only equipped with an AMD Ryzen 3700X CPU chip and an Nvidia GeForce GTX 3060 GPU card. Its CPU-GPU data transfer is completed through PCI-E. The testbed VU9P is for testing the state-of-the-art FPGA implementation. It is on Amazon EC2 using the f1.2xlarge instance, which is equipped with a Xilinx UltraScale+ VU9P FPGA card.

Table 2 gives baseline implementations that we compare in this paper. They are all state-of-the-art works achieving high efficiency on certain hardware, including GPU [Bel19, Min19, Yrr22, Mat22], CPU [Bel15], and FPGA [Har22]. PipeZK [ZWZ+21] is another recent work that provides an ASIC acceleration solution for Groth's protocol. However, its original measurements are via simulation of ASIC without a physical chip. In addition, the primary advantage of PipeZK is accelerating small-scale zkSNARK applications, while our scheme is for relatively large-scale ones. Therefore, for these different experimental setups, we do not use PipeZK as our comparator.

Note that the difference in hardware resources can significantly affect comparison results. Therefore, in order to make comparisons in a relatively fair manner, we ensure that comparisons of GPU implementations are performed in the same testbed. For comparisons between GPU and CPU implementations, we choose to perform them in the testbed G3060, where GPU/CPU chips are at a similar price. We choose to perform [Har22] on Amazon EC2 because its source code is deeply bound to this platform.

5.2 Evaluating the MSM implementation

In this section, we present the performance results of our MSM implementation. This aims to show the improvement that our parallel MSM algorithm provides exclusively. We evaluate the baseline implementations and our MSM implementation with various hardware devices. For the evaluations with our multi-GPU implementation, there are two methods for getting the final result from the subtask results. We employ the second method that transfers the subtask results from GPUs to the CPU memory. This method slightly outperforms the first method that transfers the subtask results from GPUs to a single GPU with NVLink, because the single-core performance of CPU is stronger than that of GPU with Formula (3) being executed serially. For the window size s used in our experiments, we determine it with an offline method that outputs the optimal window size s, minimizing the computational costs of our MSM algorithm. Section 4.1 provides details on this offline method and the above two methods for our multi-GPU implementation. Below we give the evaluation results.

Table 3 provides the evaluation results on systems with a single CPU/GPU/FPGA card, where the execution time is given straightforwardly and the speedup over other MSM implementations is appended below the execution time. Here, we perform these implementations with different elliptic curves due to the limitation of their optional elliptic curves. We choose the curve MNT4753 for Mina, BLS12-381 for Bellperson and Bellman, and BLS12-377 for Hardcaml. To conclude, we achieve a speedup of up to 11.44× over the CPU implementation addressed in Bellman and a speedup of up to 18.48× over the FPGA implementation addressed in Hardcaml. For the GPU implementations, our scheme has a speedup of up to 18.75× and 2.29× over Mina and Bellperson, respectively. Note that the performance of the MSM implementation addressed in Mina is relatively terrible because

Table 3: Execution time (millisecond) and speedup for MSM implementations with different MSM scales on systems with a single CPU/GPU/FPGA card.

Size	MNT4753		BLS12-381				BLS12-377	
	Mina (V100)	cuZK (V100)	Bellperson (V100)	cuZK (V100)	Bellman (3700X)	cuZK (G3060)	Hardcaml (VU9P)	cuZK (V100)
2^{19}	8701	732 (11.89×)	241	116 (2.08×)	1235	133 (9.29×)	499	27 (18.48×)
2^{20}	16071	1163 (13.82×)	409	188 (2.18×)	2391	236 (10.13×)	540	47 (11.49×)
2^{21}	31789	1960 (16.22×)	727	331 (2.20×)	4795	419 (11.44×)	620	90 (6.89×)
2^{22}	62344	3608 (17.28×)	1301	578 (2.25×)	6375	759 (8.40×)	780	171 (4.56×)
2^{23}	124429	6635 (18.75×)	2637	1154 (2.29×)	12559	1462 (8.59×)	1094	312 (3.51×)

Table 4: Execution time (millisecond) and speedup for MSM implementations with different MSM scales on systems with multiple GPUs.

Size	1 × V100		2 × V100		4 × V100		8 × V100	
	Bellper.	cuZK	Bellper.	cuZK	Bellper.	cuZK	Bellper.	cuZK
2^{20}	409	188 (2.18×)	243	101 (2.41×)	117	52 (2.25×)	62	29 (2.14×)
2^{22}	1301	578 (2.25×)	730	311 (2.35×)	415	160 (2.59×)	241	82 (2.94×)
2^{24}	5609	2059 (2.72×)	2683	1103 (2.43×)	1308	573 (2.28×)	785	297 (2.64×)
2^{26}	21772	7602 (2.86×)	11337	3977 (2.85×)	5774	2367 (2.44×)	3324	1193 (2.79×)

it employs a Straus-based parallel MSM algorithm [Str64], which cannot perform as well as Pippenger-based algorithms when the scale of MSM is large.

We also evaluate our MSM with the different number of threads to demonstrate that our MSM algorithm is adapted to the high parallelism provided by GPUs. As shown in Figure 9, the throughput of our MSM grows almost linearly with the number of threads until the thread number exceeds the GPU core number. Note that we perform the experiment in the testbed V100, where each GPU card has 5,120 cores.

Table 4 gives their execution times on systems with multiple GPUs. Here we only compare with Bellperson because it is the only baseline implementation that supports multi-GPU execution. Our MSM yields up to 2.86× (1GPUs), 2.85× (2GPUs), 2.59× (4GPUs), 2.94× (8GPUs) speedup over that in Bellperson. Here, our multi-GPU implementation shows a little non-linear acceleration with the number of GPUs. This is caused by the number of subtasks $\lceil \frac{\lambda}{s} \rceil$ (e.g., $\lceil 255/20 \rceil = 13$) being non-divisible by the number of GPUs. For example, two GPUs are assigned to 6 and 7 subtasks respectively, while four GPUs are assigned to 3, 3, 3, and 4 subtasks respectively. Therefore, it requires 7 blocks of time for 2 GPUs and 4 blocks of time for 4 GPUs (non-linear).

Next, we present the comparison results between our scheme and the two most recent GPU implementations, namely [Yrr22] and [Mat22]. These two implementations use a similar approach and both are concurrent works with our scheme. In Table 5, we show the difference and our advantages over them. First, these two implementations require a pre-computation step, while our scheme does not. The pre-computation method is used to reduce the running time by precomputing a series of EC points and treating them as other base points in MSM, as shown in [LFG23]. Note that the number of precomputed points is multiple times greater than the original base points. Therefore, storing these points in

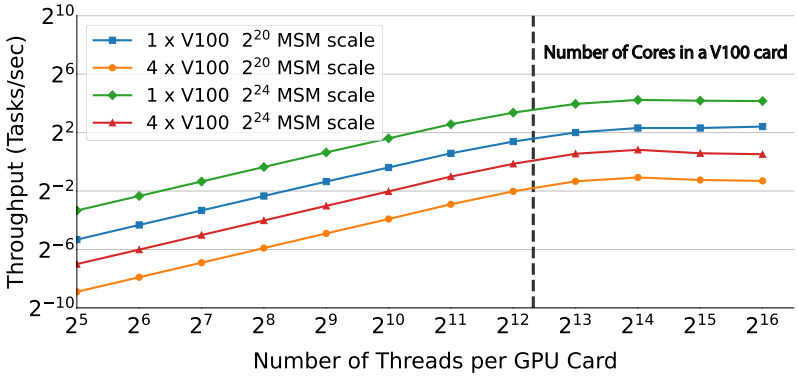

Figure 9: Throughput for our MSM scheme with the different number of execution threads.

Table 5: Execution time (millisecond) and storage cost for our MSM implementation and two concurrent works with 2^{24} MSM scale in Nvidia GeForce RTX 3090 Ti.

Imple.	Precompute time	Storage	MSM time (random scalars)	MSM time [(1)] (clustered scalars)
Yrrid	3117	6×2^{24} EC points	180	7623
MatterLab	3857	4×2^{24} EC points	205	4640
cuZK	/	1×2^{24} EC points	226	246

[(1)] Here, the MSM computation with clustered scalars represents the scalars used in MSM only have 32 different values, which are not randomly distributed.

GPU memory and utilizing them at runtime can lead to a huge storage overhead, which is an obstacle for large-scale MSM computation. Second, these two implementations focus on accelerating MSM with random scalars, rather than MSM for zkSNARK like [Gro16], where the scalars are related to the proving function inputs and are normally non-random. Their execution time for MSM with non-random scalars is much longer than ours.

5.3 Evaluating the Overall Performance of cuZK

In this section, we give the overall performance of cuZK. Here, we evaluate the baseline implementation [Bel19] and cuZK with the BLS12-381 curve and perform all experiments on the testbed V100 using single or multiple GPU cards. We omit the corresponding results of another GPU implementation Mina [Min19] for simplicity because its performance is much worse than [Bel19].

Table 6 gives the execution times of the baseline implementation and cuZK on several GPUs across various constraint scales (S). The experimental results show that cuZK has advantages in both individual operations and overall performance. It provides speedups of up to 203.59×, 2.55×, 16.06× for MUL, MSM and GPU-CPU data transfer operations, respectively. The overall performance of cuZK achieves over 2.65× (1GPU), 3.02× (2GPU), 3.53× (4GPU) speedup. Below we give a deeper insight into our experimental results.

First, we notice other GPU implementations like [Bel19] and [Min19] underestimate computational costs of the MUL operation. They choose to perform the MUL operation on CPU, which can significantly hinder the overall performance. Our approach of offloading the MUL module into GPUs greatly improves its performance. Second, the speedup of our MSM module over [Bel19] is consistent with the results described in Section 5.2. This illustrates that our MSM is well compatible with Groth's protocol. Note that there are multiple MSM operations in Groth's protocol, and their corresponding scalars are not uniformly distributed. Third, our data transfer time drops by an order of magnitude. This

Table 6: Execution time (millisecond) for Bellperson and cuZK on several GPUs across various constraint scales (S).

S	Bellperson [Bel19]				cuZK (ours)				Speedup[4]
	MUL[1]	MSM	DT[2]	Proof[3]	MUL[1]	MSM	DT[2]	Proof[3]	
1 × V100									
2^{19}	207	1903	202	2623	2.62	904	17	983	2.67
2^{20}	427	3230	417	4448	4.07	1559	29	1679	2.65
2^{21}	947	5709	881	7956	5.66	2551	56	2758	2.88
2^{22}	1846	10044	1577	14196	10.05	4687	109	5075	2.80
2^{23}	3737	20559	3358	29126	19.02	9157	209	9909	2.94
2 × V100									
2^{19}	208	1005	145	1685	2.24	478	15	555	3.03
2^{20}	422	1980	252	3086	3.94	786	24	902	3.42
2^{21}	923	3193	554	5146	5.49	1285	45	1479	3.50
2^{22}	1836	5952	1028	9499	9.97	2515	82	2875	3.30
2^{23}	3860	10357	2010	17170	18.96	4984	156	5683	3.02
4 × V100									
2^{19}	206	556	116	1896	2.89	297	24	390	4.86
2^{20}	420	877	378	2345	4.11	431	45	577	4.06
2^{21}	920	1779	667	4062	6.02	698	89	945	4.30
2^{22}	1785	3081	1129	6899	10.30	1329	153	1763	3.91
2^{23}	3831	5686	1659	12119	19.50	2579	306	3431	3.53

[1] MUL in Bellperson is executed in CPU, while that in cuZK is executed in GPU.
[2] DT represents the execution time for CPU-GPU data transfer.
[3] Proof represents the execution time for the proof generation, which consists of operations including MUL, NTT, MSM, CPU-GPU data transfer, and other less critical operations.
[4] The speedup refers to the proof generation time in Bellperson divided by the proof generation time in cuZK.

is because we overlap all MSM-required EC points transfer with the device computation, as shown in Figure 8 and Section 4.4. Fourth, our superiority over the baseline implementation becomes more apparent as the number of GPU cards increases. This benefits from that our multi-GPU implementation does not have much overhead compared to our single-GPU implementation.

Finally, we evaluate cuZK in real-world applications to demonstrate its practicality. We choose to employ cuZK in a well-known GPU-accelerated cryptocurrency application, namely Filecoin [BG17]. We modify FileCoin by extracting the constraints (represented by R1CS) used for proving correct storage in Filecoin and then using cuZK as the backend to generate the proof. We measure and compare this proof generation time using Bellperson as the default backend and using cuZK as the new backend. As a result, cuZK provides a speedup of 2.18× over the original GPU implementation of Filecoin.

6 Conclusion

Summary. In this work, we present cuZK, an efficient GPU implementation of zkSNARK. It achieves high performance with the following approaches. First, cuZK adopts a new parallel MSM algorithm. This algorithm converts the major operations used in the Pippenger algorithm to a series of basic sparse matrix operations, which leads to it adapting to the high parallelism provided by GPUs and having nearly perfect linear speedup over the Pippenger algorithm. Second, we parallelize and perform the MUL operation of zkSNARK in GPUs. Along with our self-designed MSM parallel scheme and well-studied NTT parallel scheme, cuZK achieves the parallelization of all computational

zkSNARK operations. Third, we reduce the latency overhead caused by CPU-GPU data transfer by overlapping data transfer and device computation. In addition, redundant data transfer is not needed as it is automatically eliminated in cuZK when we perform all zkSNARK operations on GPUs. As a result, our evaluation shows cuZK has a considerable speedup over other state-of-the-art GPU implementations of zkSNARK.

Further work. Our work can be extended to other ZKP protocols that require MSM, MUL, and NTT. However, it is impossible to extend our techniques to all ZKP protocols. In the future, we plan to explore more GPU-accelerated methods for a wider range of ZKP protocols. In addition, to the best of our knowledge, none of the existing CPU/GPU implementations of zkSNARK (including ours) are designed to prevent various side-channel attacks. These attacks on zkSNARK could cause information leakage. Therefore, we believe it will be a further research direction that deserves a stand-alone study.

References

[ABC+22] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. Fpga acceleration of multi-scalar multiplication: Cyclonemsm. *Cryptology ePrint Archive*, 2022.

[ABVL+20] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):379–391, 2020.

[Bai10] Eric Bainville. Opencl fast fourier transform, 2010. http://www.bealto.com/gpu-fft_group-1.html, Accessed: 2022-11-22.

[BDLO12] Daniel J Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In *International Conference on Cryptology in India*, pages 454–473. Springer, 2012.

[Bel15] Bellman: a crate for building zksnark circuits, 2015. https://github.com/zkcrypto/bellman, Accessed: 2022-11-22.

[Bel19] Bellperson: Gpu parallel acceleration for zksnark, 2019. https://github.com/filecoin-project/bellperson, Accessed: 2022-11-22.

[BG09] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.

[BG17] Juan Benet and Nicola Greco. Filecoin: A decentralized storage network. *Protocol Labs*, pages 1–36, 2017.

[BMRS20] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. *Cryptology ePrint Archive*, 2020.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–768. Springer, 2020.

[CL03] Chin-Chen Chang and Der-Chyuan Lou. Fast parallel computation of multi-exponentiation for public key cryptosystems. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 955–958. IEEE, 2003.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[DBOG14] S Dalton, N Bell, L Olson, and M Garland. Cusp: A c++ templated sparse matrix library., 2014. https://cusplibrary.github.io, Accessed: 2022-11-22.

[Gar08] Michael Garland. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th annual design automation conference*, pages 2–6, 2008.

[GD14] Joseph L Greathouse and Mayank Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780. IEEE, 2014.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[GJCC20] Naina Gupta, Arpan Jati, Amit Kumar Chauhan, and Anupam Chattopadhyay. Pqc acceleration using gpus: Frodokem, newhope, and kyber. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):575–586, 2020.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Annual International Cryptology Conference*, pages 581–612. Springer, 2017.

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

[Gna20] gnark-crypto: gnark-crypto provides efficient cryptographic primitives in go., 2020. https://github.com/ConsenSys/gnark-crypto.git, Accessed: 2022-11-22.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 321–340. Springer, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[GXW21] Yiwen Gao, Jia Xu, and Hongbing Wang. Cunh: Efficient gpu implementations of post-quantum kem newhope. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):551–568, 2021.

[Har22] Hardcaml zprize, 2022. https://zprize.hardcaml.com/, Accessed: 2022-12-09.

[HB22] Youssef EL Housni and Gautam Botrel. Edmsm: Multi-scalar-multiplication for recursive snarks and more. *Cryptology ePrint Archive*, 2022.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732, 1992.

[KJPA20] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 264–275. IEEE, 2020.

[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*, pages 358–380, 2023.

[Lib14] libsnark: a c++ library for zksnark proofs, 2014. https://github.com/scipr-lab/libsnark, Accessed: 2022-11-22.

[LZW21] Gangzhao Lu, Weizhe Zhang, and Zheng Wang. Optimizing depthwise separable convolution operations on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):70–87, 2021.

[Mat22] Accelerating msm operations on gpu/fpga, 2022. https://github.com/matter-labs/z-prize-msm-gpu, Accessed: 2022-12-09.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[Mil04] Victor S Miller. The weil pairing, and its efficient calculation. *Journal of cryptology*, 17(4):235–261, 2004.

[Min19] Mina: gpu groth16 prover, 2019. https://github.com/MinaProtocol/gpu-groth16-prover-3x, Accessed: 2022-11-22.

[NCVK10] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[Opp99] Alan V Oppenheim. *Discrete-time signal processing*. Pearson Education India, 1999.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 258–263. IEEE Computer Society, 1976.

[Pol22] Polygon: Ethereum transparent scalability with l2 zk-rollup., 2022. https://polygon.technology/solutions/polygon-zkevm, Accessed: 2022-11-22.

[Roo94] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 389–399. Springer, 1994.

[SCG+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. IEEE, 2014.

[Spp22] Zero-knowledge template library., 2022. https://github.com/supranational/sppark, Accessed: 2022-12-09.

[Str64] Ernst G Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70(806-808):16, 1964.

[TDM+14] Yuan Tao, Yangdong Deng, Shuai Mu, Mingfa Zhu, Limin Xiao, Li Ruan, and Zhibin Huang. Atomic reduction based sparse matrix-transpose vector multiplication on gpus. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 987–992. IEEE, 2014.

[WQS+20] Xinlei Wang, Yuxing Qiu, Stuart R Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. A massively parallel and scalable multi-gpu material point method. *ACM Transactions on Graphics (TOG)*, 39(4):30–1, 2020.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. Dizk: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[Xav22] Charles F Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. *Cryptology ePrint Archive*, 2022.

[Yrr22] Z-prize msm on the gpu submission., 2022. https://github.com/yrrid/submission-msm-gpu, Accessed: 2022-12-09.

[ZC15] Zhichao Zhao and T-H Hubert Chan. How to vote privately using bitcoin. In *International Conference on Information and Communications Security*, pages 82–96. Springer, 2015.

[ZFZS20] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2039–2053, 2020.

[ZGK+17] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880. IEEE, 2017.

[Zpr22] Accelerating msm operations on gpu., 2022. https://www.zprize.io/prizes/accelerating-msm-operations-on-gpu-fpga, Accessed: 2022-12-09.

[ZWW+21] Lingchen Zhao, Qian Wang, Cong Wang, Qi Li, Chao Shen, and Bo Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2524–2540, 2021.

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