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

Tao Lu, Chengkun Wei, Ruijing Yu, Yi Chen, Li Wang, Chaochao Chen, Zeke Wang, and Wenzhi Chen

Abstract—Zero-knowledge proof (ZKP) is a critical cryptographic protocol, and it has been deployed in various privacy-preserving applications such as cryptocurrencies and verifiable machine learning. Unfortunately, ZKP 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) on elliptic curves. Currently, several GPU-accelerated implementations of ZKP have been developed to improve its performance. However, these existing GPU designs do not fully unleash the potential of GPUs. Therefore, this paper presents cuZK, an efficient GPU implementation of ZKP with the following three optimizations to achieve higher performance. First, we propose a new parallel MSM algorithm and deploy it in cuZK. This MSM algorithm is well adapted to the high parallelism provided by GPUs, and it achieves nearly perfect linear speedup over the Pippenger algorithm, a well-known serial MSM algorithm. Second, we parallelize the MUL operation, which is lightly disregarded by other existing GPU designs. Indeed, along with our self-designed MSM parallel scheme and well-studied NTT parallel 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 (DT) by 1) reducing redundant data transfer and 2) overlapping data transfer and device computation with the multi-streaming technique. We design a series of evaluation schemes for cuZK. The evaluation results show that our MSM module provides over $2.08\times$ (up to $2.63\times$) speedup versus the state-of-the-art GPU implementation. cuZK achieves over $2.65\times$ (up to $4.47\times$) speedup on standard benchmarks and $2.18\times$ speedup on a GPU-accelerated cryptocurrency application, Filecoin.

Index Terms—Zero-knowledge proof, parallel algorithm, multi-scalar multiplication, cryptographic, zkSNARK, GPU.

1 Introduction

Zero-knowledge proof (ZKP) [1] is a cryptographic protocol that allows a *prover* to generate a proof π to convince *verifier*s that a computation $y = f(x, w)$ is correctly calculated with a public input x and a prover's secret input w. 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 [2], which stands for *zero-knowledge Succinct Non-interactive ARgument of Knowledge*. Compared with other traditional ZKPs [3], [4], [5], zkSNARK has much more succinct proof π, which 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 [6], verifiable database outsourcing [7], cryptocurrencies [8], [9], [10], [11], and verifiable machine learning [12], [13].

However, there is still a bottleneck that limits further deployments of zkSNARK. Currently, the state-of-the-art zkSNARK [2] (also traditional ZKPs) has a high overhead on its proof generation step. To generate a proof, the prover has to perform various time-consuming operations, including 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 the solutions to reduce the proof generation time is parallelizing this task on certain hardware. GPUs are many-core computing platforms that support 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 [14], [15], cryptography [16], [17], and graphics [18]. There are also several existing GPU designs of zkSNARK. For example, Mina announced a challenge for speeding up zkSNARK using GPUs with a high reward ($ 100k). The final acceleration result of this challenge has been open-sourced in [19]. Another GPU implementation Bellperson [20] is improved from a CPU-based version Bellman [21]. Bellperson has been deployed in a well-known decentralized cryptocurrency network Filecoin [10]. Figure 1 shows the percentage of their execution time on zkSNARK operations, including MUL, NTT, MSM, and the GPU-CPU data transfer (DT). Obviously, the overall performance of zkSNARK largely depends on the efficiency of the above four operations. Especially, MSM is the most time-consuming operation, taking more than 70 percent of

• *Tao Lu, Chengkun Wei, Ruijing Yu, Yi Chen, Chaochao Chen, Zeke Wang, and Wenzhi Chen are with Zhejiang Univerisity.*
E-mail: {lutao2020, weichengkun, rjyu, chenyi2000, zjuccc, wangzeke, chenwz}@zju.edu.cn
Li Wang is with Ant Financial Group.
E-mail: raymond.wangl@antgroup.com

the total runtime.

Nevertheless, the zkSNARK operations performed in existing GPU implementations have the following three weaknesses. **1) MSM:** Their parallel algorithms for the MSM computation are simply modified from those 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:** Existing GPU designs perform the MUL operation serially on a CPU rather than parallelly on GPUs. The reason for choosing such a design scheme may be due to the matrix that MUL operates on being too large to be stored in the GPU memory directly. The slow way of running MUL serially hinders the overall performance; **3) DT:** 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 [19] consumes much time in performing NTT serially, but we still do not consider NTT to be a weakness of existing zkSNARK implementations, as there are already many efficient parallel NTT schemes [16], [22], [23]. We can easily replace the serial scheme with a parallel one. For example, the parallel NTT scheme used in Bellperson is actually from OpenCL [24].

In this paper, we present cuZK, an efficient GPU implementation of zkSNARK. We make three optimizations to help cuZK achieve higher performance. First, we propose a new parallel MSM algorithm. This algorithm is not only well adapted to the high parallelism provided by GPUs, but also has lower computational costs than existing MSM parallel algorithms. Therefore, we deploy this new algorithm in cuZK's MSM module. Second, we notice the matrix that MUL operates on is very large but sparse. We represent it in sparse matrix format, which allows us to store the whole matrix on GPUs and to perform MUL computation with parallel schemes on sparse matrices. Furthermore, along with our self-designed MSM parallel scheme and well-studied NTT parallel scheme [16], [22], [23], cuZK indeed achieves the parallelization of all zkSNARK operations. Third, cuZK reduces the latency overhead caused by CPU-GPU data transfer by overlapping data transfer and device computation using the multi-streaming technique. Note that redundant data transfer is automatically eliminated after we perform all zkSNARK operations on GPUs. To sum up, cuZK achieves high performance by optimizing three critical zkSNAKR operations, namely MSM, MUL, and DT.

In particular, our proposed parallel MSM algorithm has the greatest impact on the overall performance improvement of cuZK. This MSM algorithm is unlike other traditional 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. Then, we convert the major operations used in the Pippenger algorithm [25], a well-known serial MSM algorithm, to a series of basic sparse matrix operations, including sparse matrix transpose and sparse matrix-vector multiplication. This enables us to utilize the technologies used in well-studied parallel sparse matrix algorithms [26], [27], [28] to accelerate the MSM computation. As a result, our parallel MSM algorithm has nearly perfect linear speedup over the Pippenger algorithm, where perfect linear speedup means the speedup ratio is equal to the number of execution threads.

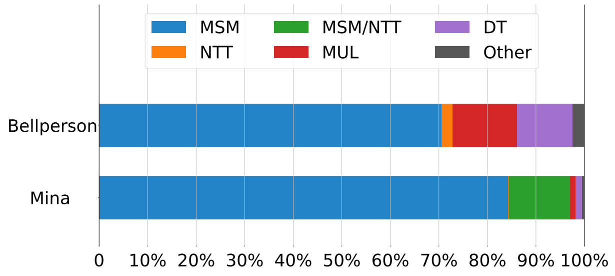


Fig. 1: The percentage of the execution time on zkSNARK operations. The label MSM/NTT means that MSM and NTT are executed simultaneously.

Precisely, the main contributions and scope of the paper can be summarized as follows:

- We propose a new parallel MSM algorithm. This algorithm is well adapted to the high parallelism provided by GPUs, and it has nearly perfect linear speedup over the Pippenger algorithm.
- We present cuZK, an efficient GPU implementation of zkSNARK. It achieves high performance by deploying a faster MSM algorithm, parallelizing the MUL operation, offloading all zkSNARK operations to GPUs, and overlapping CPU-GPU data transfer and device computation.
- We design a series of evaluation schemes for cuZK. The evaluation results show that our MSM module provides over $2.08\times$ (up to $2.63\times$) speedup versus the state-of-the-art GPU implementation. The overall performance of cuZK achieves over $2.65\times$ (up to $4.47\times$) speedup on standard benchmarks and $2.18\times$ speedup on a GPU-accelerated cryptocurrency application, Filecoin.

The rest of the paper is organized as follows: In Section 2, we introduce the preliminaries for this paper. In Section 3, we provide full details of cuZK. Additionally, our proposed MSM parallel algorithm is also described here. Section 4 gives our experiments, benchmarking, and comparison results. Finally, Section 5 draws some conclusions and provides guidelines for potential future work.

2 Preliminaries

In this section, we give an introduction to the zkSNARK protocol, multi-scalar multiplication, the Pippenger algorithm, sparse matrix, as well as Graphics Processing Units.

2.1 The zkSNARK Protocol

The zkSNARK protocol [2] is one of the state-of-the-art ZKPs. It works like all ZKPs that allow 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. The proof π leaks no information about the secret input w. Compared with other traditional ZKPs [3], [4], [5], the advantage of zkSNARK is its much more

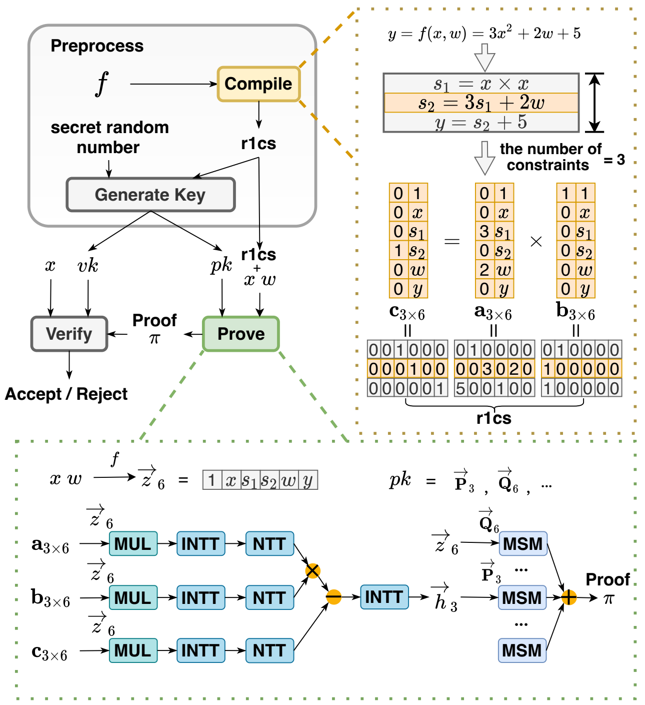


Fig. 2: The workflow of the zkSNARK protocol. INTT is the inverse transformation of NTT. Its operation is similar to NTT.

succinct proof, which 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 adopted by various private-preserving applications, including electronic voting [6], verifiable database outsourcing [7], cryptocurrencies [8], [9], [10], and verifiable machine learning [12], [13].

The workflow of zkSNARK 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, due to the proof π leaking no information about the secret input w, no one except the owner can get the value of the secret input w.

In zkSNARK protocol, the `preprocess` procedure and the `verify` procedure have lightweight computational

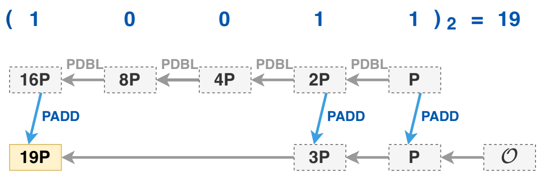


Fig. 3: An example of PMUT computation. $\mathcal{O}$ is the zero element on elliptic curve.

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 [29], which only takes several milliseconds. The `prove` procedure is the only high-expense procedure in zkSNARK. As shown at the bottom 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; see 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 $\mathbf{P}_i$. In short, EC points are basic operands in elliptic curve arithmetic. They supports several basic arithmetic operations such as point addition (PADD) and point double (PDBL). 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 [30] to compute the pair $k\mathbf{P}$ is by performing a series of PDBLs and PADDs. Figure 3 shows an example of computing $19\mathbf{P}$. We start with representing the scalar 19 in the binary form $(10011)_2$. Then, at each bit position, we double the point $\mathbf{P}$ and add it to the result when the bit is 1. The EC point obtained 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, λ commonly ranges from 254 to 768. The scale of MSM n could be extremely large. For instance, Filecoin [10] 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 [25], the Bos-Coster algorithm [31], and the Chang-Lou algorithm [32]. Especially, the Pippenger algorithm performs best when the scale of MSM is very large; see [33].

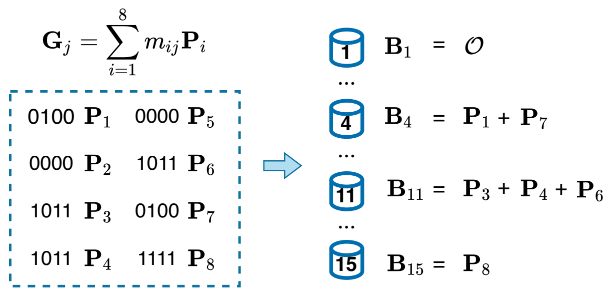


Fig. 4: An example of putting EC points into buckets.

2.3 The Pippenger Algorithm

The Pippenger algorithm [25] is a popular serial algorithm for the MSM computation. Our proposed parallel MSM algorithm is inspired by it. Therefore, in this section, we first review the Pippenger algorithm and analyze its computational costs. Then, we briefly introduce three existing parallel Pippenger-based algorithms. The details of the Pippenger algorithm are shown in Algorithm 1, which mainly consists of three steps.

The first step is to convert the original task $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$ to multiple smaller subtasks. More specifically, it starts by choosing a window size s and then 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).

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

The second step is to 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 k_i into a bucket whose index is equal to k_i. 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 $l \in [1, 2^s - 1]$. We can find 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 [33] to compute $\sum_{l=1}^{2^s-1} l\mathbf{B}_l$. Details are presented in Algorithm 2. In short, it starts by calculating a serial of new EC 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 EC points $\mathbf{M}_l$, as shown in Formula (2). Actually, Formula (2) achieves converting the expensive PMULT to the lightweight PADD.

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

Algorithm 1 The Pippenger Algorithm [25]

```
Require: A scalar vector $\overrightarrow{k}_n = [k_1, k_2, ..., k_n]$, whose
    elements are $\lambda$-bit scalars. A point vector $\overrightarrow{\mathbf{P}}_n =$
    $[\mathbf{P}_1, \mathbf{P}_2, ..., \mathbf{P}_n]$. A chosen window size $s$.
Ensure: $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$
 1: // Convert the original task into $\lceil \frac{\lambda}{s} \rceil$ subtasks.
 2: for $j \leftarrow (\lceil \frac{\lambda}{s} \rceil)$ to 1 do
 3:     // Empty $2^s - 1$ buckets, $\mathcal{O}$ is the zero element on EC.
 4:     $\overrightarrow{\mathbf{B}}_{2^s-1} \leftarrow [\mathcal{O}, \mathcal{O}, ..., \mathcal{O}]_{2^s-1}$.
 5:     // Put each $\mathbf{P}_i$ into the corresponding bucket and add up
    all points in the same bucket.
 6:     for $i \leftarrow 1$ to $n$ do
 7:         // $m_{ij}$ is a part of $k_i$ used in this subtask.
 8:         $m_{ij} \leftarrow (k_i \gg ((j-1) * s)) \,\&\, ((1 \ll s) - 1)$
 9:         if $m_{ij} \neq 0$ then
10:             $\mathbf{B}_{m_{ij}} \leftarrow \mathbf{B}_{m_{ij}} + \mathbf{P}_i$
11:         end if
12:     end for
13:     // Get the result of this subtask, $\mathbf{G}_j = \sum_{l=1}^{2^s-1} l\mathbf{B}_l$.
14:     $\mathbf{G}_j \leftarrow$ BucketPointsReduction($\overrightarrow{\mathbf{B}}_{2^s-1}$)
15:     // Add $\mathbf{G}_j$ to the final result based on Formula (3).
16:     if $j \neq \lceil \frac{\lambda}{s} \rceil$ then
17:         $\mathbf{T}_j \leftarrow 2^s \mathbf{T}_{j+1} + \mathbf{G}_j$
18:     else
19:         $\mathbf{T}_j \leftarrow \mathbf{G}_j$
20:     end if
21: end for
22: $\mathbf{Q} = \mathbf{T}_1$
23: return $\mathbf{Q}$
```

Algorithm 2 BucketPointsReduction [33]

```
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: // $\mathcal{O}$ is the zero element on EC.
 2: $\mathbf{G} \leftarrow \mathcal{O}$
 3: for $l \leftarrow 1$ to $2^s - 1$ do
 4:     if $l \neq 1$ then
 5:         $\mathbf{M}_l \leftarrow \mathbf{M}_{l-1} + \mathbf{B}_{2^s-l}$
 6:     else
 7:         $\mathbf{M}_l \leftarrow \mathbf{B}_{2^s-1}$
 8:     end if
 9: end for
10: for $l \leftarrow 1$ to $2^s - 1$ do
11:     $\mathbf{G} \leftarrow \mathbf{G} + \mathbf{M}_l$
12: end for
13: return $\mathbf{G}$
```

The third step is to compute the MSM result with the subtask results, namely $\mathbf{Q} = \sum_{j=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(j-1)s} \mathbf{G}_j$. It also starts by calculating a serial of new EC 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 given by 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, we can find 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 $(2^{s+1} - 3)$ 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.

2.3.1 Existing Parallel Pippenger-based Algorithms

Thanks to the outstanding performance of the Pippenger algorithm, many state-of-the-art parallel MSM algorithms used in zkSNARK are based on it. Here, we briefly introduce three existing parallel Pippenger-based algorithms and then give their computational costs.

The first algorithm exists in a CPU implementation of zkSNARK, gsnark [34]. 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, where $\lceil\frac{\lambda}{s}\rceil$ is the total number of subtasks. After all threads obtain the subtask results, one of the threads adds these results to the final result based on Formula (3). This parallel algorithm provides a speedup of at most $\lceil\frac{\lambda}{s}\rceil$, 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 algorithm is a more general parallel algorithm of MSM. It exists in many zkSNARK implementations, including Bellman [21], libsnark [35], and DIZK [36]. This algorithm 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 algorithm combines the above two parallel algorithms. It exists in a GPU implementation, Bellperson [20]. 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 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.

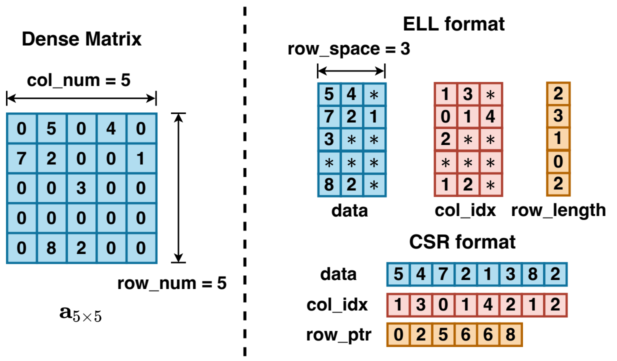


Fig. 5: Sparse matrix representations for a simple example matrix $\mathbf{a}_{5\times5}$.

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.

Compressed sparse row (CSR) format [37] and ELLPACK (ELL) format [38] 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`, `col_idx`, and `row_length`. Specifically, the nonzero elements in the same row of the sparse matrix are stored in the same row of the `data`. The `col_idx` stores the column indices of these nonzero elements. All rows of the `data` and `col_idx` structures are padded to length `row_space` to meet the alignment requirement. The `row_length` stores the number of the nonzero elements in each row of the sparse matrix. The CSR format also consists of three structures, `data`, `col_idx`, and `row_ptr`. 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 `row_ptr` is an array of length $\texttt{row\_num} + 1$. Its i-th element encodes the cumulative number of nonzero elements up to the i-th row, where $i \in [0, \texttt{row\_num}]$.

The basic operations supported by sparse matrices include sparse matrix transposition, sparse matrix-vector multiplication, and so on. Many well-studied GPU implementations [26], [27], [28] 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 [39], [40]. Therefore, converting other complex operations to basic sparse matrix operations is commonly a suitable and convenient choice to improve the efficiency of the computation [41], [42], [43].

2.5 Graphics Processing Units

Graphics Processing Units (GPUs) are many-core computing platforms that support 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 typically consists of 32 or 64 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 Our GPU Implementation

In this section, we present cuZK, our efficient GPU implementation of zkSNARK. We first introduce our design challenges and goals, and then provide full details of our implementation.

3.1 Design Challenges and Goals

As mentioned in Section 2.1, the `prove` procedure is the most time-consuming part of zkSNARK. It requires the prover to generate a proof by performing a series of operations, including large-scale matrix-vector multiplication (MUL), number-theoretic transform (NTT), and multi-scalar multiplication (MSM) on elliptic curves. Our work focuses on improving the performance of this part with GPUs. However, promoting the efficiency of proof generation is not something that can be achieved painlessly. For example, several GPU designs [19], [20] have challenged this task, but their failure to fully utilize the computing power of GPUs makes the final acceleration results unsatisfactory. Here, we pose three design challenges according to the weaknesses addressed in the existing GPU designs.

1) *There is no parallel MSM algorithm adapted to the high parallelism provided by GPUs.* The algorithms addressed in the existing GPU designs are simply modified from those used in the low-parallelism setting, which manifests an unsatisfiable increase in speedup with the increasing parallelism.
2) *The matrices that MUL operates on are too large to be stored in the GPU memory directly.* Thus, the existing GPU designs choose to perform the MUL operation serially in a CPU, which hinders the overall performance of zkSNARK.
3) *The latency overhead caused by large blocks of CPU-GPU data transfer results in performance degradation.* On the one hand, the existing GPU designs have a great amount of redundant data transfer; see Figure 6(a). On the other hand, the prover key, which consists of multiple large-scale EC point vectors, is required to be moved to GPUs. It is very large in size.

In response to these challenges, we set the following four design goals for our implementation.

1) *Deploying a new faster parallel MSM algorithm.* This parallel MSM algorithm is not only required to be adapted to high parallelism provided by GPUs but also needs to have lower computational costs than existing MSM parallel algorithms.
2) *Parallelizing the MUL operation.* The MUL operation performs the matrix-vector multiplication on an R1CS matrix. Therefore, we should choose an appropriate parallel scheme for the MUL computation based on the characteristics of the R1CS matrix.
3) *Offloading all zkSNARK operations to GPUs.* With our self-designed MSM and MUL parallel schemes, as well as the well-studied NTT parallel scheme, we indeed can achieve the parallelization of all zkSNARK operations. These parallel schemes should be well-suitable for execution on GPUs.
4) *Reducing the latency overhead caused by CPU-GPU data transfer.* On the one hand, redundant data transfer should be eliminated after we perform all zkSNARK operations on GPUs; see Figure 6(b). On the other hand, a fine-grained scheme of overlapping data transfer and device computation needs to be designed for large blocks of CPU-GPU data transfer.

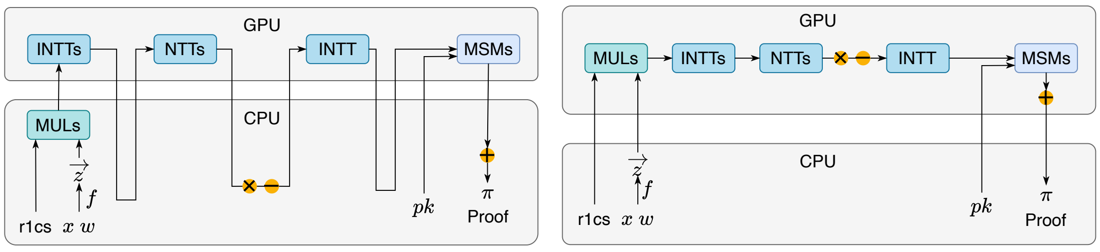
3.2 Multi-scalar Multiplication

Multi-scalar multiplication (MSM) is the most time-consuming operation in zkSNARK, taking more than 70 percent of the total runtime. Its definition is given by the formula $\mathbf{Q} = \overrightarrow{k}_n \cdot \overrightarrow{\mathbf{P}}_n = \sum_{i=1}^{n} k_i \mathbf{P}_i$, where n is the scale of MSM, k_i is a λ-bits scalar, $\mathbf{P}_i$ is an EC point, and the pair $k_i \mathbf{P}_i$ represents PMULT of k_i and $\mathbf{P}_i$. More details of the MSM operation are shown in Section 2.2.

In this section, we first introduce our proposed parallel MSM algorithm (Section 3.2.1) and then present the essential details of its deployment on GPUs (Section 3.2.2).

3.2.1 Parallel MSM Algorithm

Our proposed parallel MSM algorithm is inspired by the Pippenger algorithm [25], which is a famous serial algorithm for the MSM computation. The details of the Pippenger algorithm are stated in Section 2.3. Our algorithm is well-suitable for execution in GPUs and has nearly perfect linear speedup over the Pippenger algorithm. The details of



(a) Bellperson (b) Ours

Fig. 6: The workflows of Bellperson and ours. In Bellperson, only NTT and MSM are performed on GPUs, thus requiring a great deal of redundant CPU-GPU data transfer. As a contrast, in our implementation, all zkSNARK operations are performed on GPUs, and redundant CPU-GPU data transfer is no longer required.

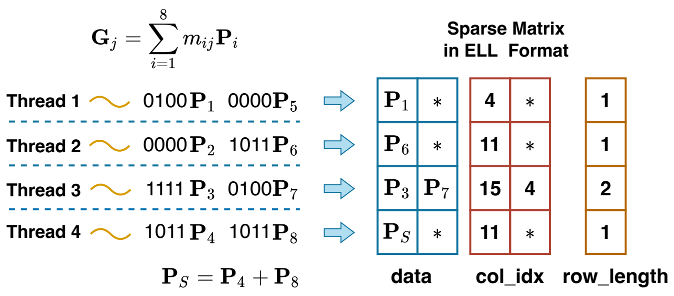

$\mathbf{G}_j = \sum_{i=1}^{8} m_{ij}\mathbf{P}_i$
Sparse Matrix in ELL Format
Thread 1 0100$\mathbf{P}_1$ 0000$\mathbf{P}_5$
Thread 2 0000$\mathbf{P}_2$ 1011$\mathbf{P}_6$
Thread 3 1111 $\mathbf{P}_3$ 0100$\mathbf{P}_7$
Thread 4 1011$\mathbf{P}_4$ 1011$\mathbf{P}_8$
$\mathbf{P}_1$ * 4 * 1
$\mathbf{P}_6$ * 11 * 1
$\mathbf{P}_3$ $\mathbf{P}_7$ 15 4 2
$\mathbf{P}_S$ * 11 * 1
$\mathbf{P}_S = \mathbf{P}_4 + \mathbf{P}_8$
data
col_idx
row_length

Fig. 7: An example of putting EC points into sparse matrix in parallel.

our parallel MSM algorithm are described below and shown in Algorithm 3.

Similar to 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:

The first step is to 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 `row_space` is $\frac{n}{t}$. Then, we launch t threads to store these EC points into the sparse matrix in parallel. Specifically, as shown in Figure 7, 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 directly.

The second step is to 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. We begin with converting the sparse matrix in ELL format to CSR format and then transpose it in parallel. The reason that we employ the CSR format is to save space costs, since the alignment requirement of the ELL format leads to additional space overhead for storing the matrix. Next, we add up all EC

Algorithm 3 Our Parallel MSM Algorithm

Require: A scalar vector $\vec{k}_n = [k_1, k_2, ..., k_n]$, 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.

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

```
1:  // Convert the original task into $\lceil \frac{\lambda}{s} \rceil$ subtasks.
2:  for $j \leftarrow 1$ to $\lceil \frac{\lambda}{s} \rceil$ do
3:      // Generate an empty sparse matrix in ELL format.
4:      $row\_num \leftarrow t$
5:      $col\_num \leftarrow 2^s - 1$
6:      $row\_space \leftarrow \frac{n}{t}$
7:      $ell \leftarrow \mathsf{GenELLMtx}(row\_num, col\_num, row\_space)$
8:      // $m_i$ is a part of $k_i$ used in this subtask
9:      for $i \leftarrow 1$ to $n$ do in parallel
10:         $m_i \leftarrow (k_i \gg ((j-1) * s))\ \&\ ((1 \ll s) - 1)$
11:     end for
12:     Synchronize()
13:     $\vec{m}_n \leftarrow [m_1, m_2, ..., m_n]$
14:     // Store EC points into the sparse matrix; see Figure 7.
15:     $ell \leftarrow \mathsf{pStoreECPoints}(ell, \vec{m}_n, \overrightarrow{\mathbf{P}}_n, t)$
16:     $csr \leftarrow \mathsf{pELL2CSR}(ell, t)$
17:     $csr \leftarrow \mathsf{pTranspose}(csr, t)$
18:     // A scalar vector whose elements are all equal to 1.
19:     $\vec{v}_t \leftarrow [1, 1, ..., 1]_t$
20:     $\overrightarrow{\mathbf{B}}^{\langle j \rangle}_{2^s-1} \leftarrow \mathsf{pSparseMatrixVectorMUL}(csr, \vec{v}_t, t)$
21: end for
22: for $j \leftarrow 1$ to $\lceil \frac{\lambda}{s} \rceil$ do in parallel
23:     $\mathbf{G}_j \leftarrow \mathsf{pBucketPointsReduction}(\overrightarrow{\mathbf{B}}^{\langle j \rangle}_{2^s-1}, t / \lceil \frac{\lambda}{s} \rceil)$
24: end for
25: Synchronize()
26: // Add $\mathbf{G}_j$ to the final result with Fomurla (3).
27: $\mathbf{T}_{\lceil \frac{\lambda}{s} \rceil} \leftarrow \mathbf{G}_{\lceil \frac{\lambda}{s} \rceil}$
28: for $j \leftarrow (\lceil \frac{\lambda}{s} \rceil - 1)$ to 1 do
29:     $\mathbf{T}_j \leftarrow 2^s\mathbf{T}_{j+1} + \mathbf{G}_j$
30: end for
31: $\mathbf{Q} \leftarrow \mathbf{T}_1$
32: return $\mathbf{Q}$
```

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 $t$.
Ensure: An EC point $\mathbf{G} = \sum_{l=1}^{2^s-1} l\mathbf{B}_l$.
 1: // Thread ID, $\xi \in [1, t]$.
 2: $\xi \leftarrow$ GetThreadID()
 3: // Divide $2^s - 1$ vector elements into $t$ parts. Each part has $r$
    EC points.
 4: $r \leftarrow (2^s - 1)/t$
 5: for $l \leftarrow 1$ to $r$ do
 6:     if $l \neq 1$ then
 7:         $\mathbf{M}_{(\xi-1)r+l} \leftarrow \mathbf{M}_{(\xi-1)r+l-1} + \mathbf{B}_{\xi r+1-l}$
 8:         $\mathbf{S}_\xi \leftarrow \mathbf{S}_\xi + \mathbf{M}_{(\xi-1)r+l}$
 9:     else
10:         $\mathbf{M}_{(\xi-1)r+l} \leftarrow \mathbf{B}_{\xi r}$
11:         $\mathbf{S}_\xi \leftarrow \mathbf{M}_{(\xi-1)r+l}$
12:     end if
13: end for
14: // After completing the above loop,
15: // $\mathbf{S}_\xi = \mathbf{B}_{(\xi-1)r+1} + 2\mathbf{B}_{(\xi-1)r+2} + ... + r\mathbf{B}_{(\xi-1)r+r}$
16: // $\mathbf{M}_{\xi r} = \mathbf{B}_{(\xi-1)r+1} + \mathbf{B}_{(\xi-1)r+2} + ... + \mathbf{B}_{(\xi-1)r+r}$
17: $\mathbf{S}_\xi \leftarrow \mathbf{S}_\xi + ((\xi - 1)r)\mathbf{M}_{\xi r}$
18: Synchronize()
19: $\overrightarrow{\mathbf{S}}_t \leftarrow [\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_t]$
20: $\mathbf{G} \leftarrow$ pSum($\overrightarrow{\mathbf{S}}_t, t$)
21: // After completing the pSum function,
22: // $\mathbf{G} = \mathbf{S}_1 + \mathbf{S}_2 + ... + \mathbf{S}_t$
23: return $\mathbf{G}$
```

points that are in the same row of the transposed matrix, which is equivalent to performing the sparse matrix-vector multiplication (SPMV) on the matrix and a scalar vector whose elements are all equal to 1. The SPMV result is the EC point vector that we need. Note that the above sparse matrix operations are all performed in parallel with t threads.

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. We can find the subtask results are exactly equal to the results of Algorithm 4, 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).

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 SPMV method cannot guarantee the workload of each thread is the same. Fortunately, this problem can be mitigated with our proposed SPMV approach; see Section 3.2.2. 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.

3.2.2 Implementation of Parallel MSM on GPUs

Our GPU implementation of MSM is based on the parallel MSM algorithm proposed in Section 3.2.1. Below, we present some crucial parts of our GPU implementation in detail.

We start by generating a sparse matrix of ELL format on the global memory so that every thread can access this matrix. Then, each thread should have performed as in Figure 7 to store EC points. However, in practice, what we store in the sparse matrix is not the EC points themselves but their indexes in the EC point vector. This step helps to save device storage costs. Roughly, each EC point typically has hundreds of bits, while the index size is the logarithm of the vector scale, only tens of bits. Next, after the indexes of these EC points are stored into the sparse matrix, we convert this matrix to the CSR format and then transpose it in parallel.

Afterward, we fetch the corresponding EC points from host memory to device memory according to the indices stored in the matrix and then sum up the EC points that are in the same row of the transposed 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; see Section 3.5 for details. The summation step is actually equivalent to performing parallel sparse matrix-vector multiplication (SPMV) on the matrix and 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 warps instead of threads to work across these groups, and thus all threads in a 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 is 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.

Another crucial part we need to be concerned with is the parallel sum operation used in Algorithm 4. In fact, it is a basic reduction operation commonly used in parallel programming to add up all elements of a vector. The CUB library [44] provides GPU implementation of the parallel sum operation.

Finally, we present the multi-GPU implementation of our MSM algorithm. As described in section 3.2.1, our parallel MSM algorithm decomposes the original MSM task into multiple subtasks. Here, we assign these subtasks to GPUs evenly. Specifically, each GPU allocates its global memory for a sparse matrix and follows the operations described in the above four paragraphs to complete its corresponding subtasks. There is no requirement for any GPU-GPU data transfer, except that we need eventually add the subtask results to the final result via Formula (3). Note that each subtask result is an EC point. Therefore, our multi-GPU implementation does not introduce substantial additional overhead compared to our single-GPU implementation.

3.3 Matrix-vector Multiplication

In zkSNARK, matrix-vector multiplication (MUL) operates on the R1CS matrix that is compiled from the function to be proved in zkSNARK; see 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. 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 matrices. There are many parallel sparse matrix-vector multiplication (SPMV) schemes, including CSR-Scalar [45], CSR-Vector [26], and CSR-Balanced. However, these schemes are not suitable for all 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. The characteristics of the R1CS matrix are not fixed. They depend on the function to be proved in zkSNARK, as shown in Section 2.1. 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 matrix characteristics calculation and the sort operation existing in CSR-Balanced can actually be performed offline because the R1CS matrix that MUL operates on is infinitely reusable for the function to be proved in zkSNARK; see Section 2.1. Therefore, this method does not introduce additional overhead for the MUL computation online.

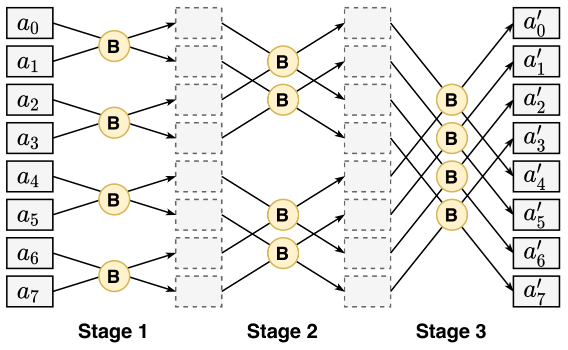


Fig. 8: The butterfly diagram for an 8-point NTT. B represents the butterfly operation.

3.4 Number-theoretic Transform

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 $\overrightarrow{a}'_N \overset{\text{def}}{=} \mathsf{NTT}(\overrightarrow{a}_N)$ 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 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 [16], [23] that have been developed for its computation. For instance, a state-of-the-art implementation can be found in [22], which is 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 [46], we decomposes the overall computation of NTT into $\log N$ stages, where each stage requires $N/2$ *butterfly* operations [47]. A single butterfly operation performs reading two input values, processing input values, and storing results. Figure 8 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.

3.5 Workflow of cuZK

Actually, we have achieved the parallelization of all zkSNARK operations with our self-designed MSM and MUL parallel schemes and the well-studied NTT parallel scheme. Moreover, these parallel schemes are well-suitable for execution on GPUs. Therefore, to make the best use of these parallel schemes, we offload all zkSNARK operations to GPUs.

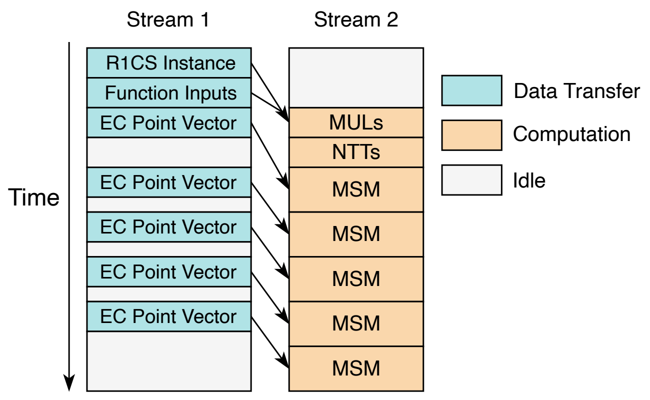


Fig. 9: Timeline for the execution of cuZK with two streams.

The overall workflow of cuZK is shown in Figure 6(b). Note that redundant CPU-GPU data transfer is no longer required after all zkSNARK operations are performed on GPUs. Only three storage modules need to be sent to GPUs once, namely the R1CS instance, the function inputs, and the prover key.

In detail, we first transmit the R1CS instance and the function inputs. As stated in Section 2.1 and Section 3.3, 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. In addition, we must finish this transfer before the device computation begins because the above two storage modules are required by the MUL operation, which is the first performed operation in the proof generation step of zkSNAKR.

Another essential storage module, the prover key, consists of multiple large-scale EC point vectors and thus is very large 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 9, we transmit the first MSM-required EC point vector while executing MULs and NTTs, the second MSM-required EC point vector while executing the first MSM operation, and so on. 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.

4 Evaluation

The evaluation results presented in this section consist of three parts. First, we give the benchmark results for our MSM implementation. This aims to show the improvement that our parallel MSM algorithm provides exclusively. Second, we present the overall performance of cuZK. Third, various real-world applications are showcased to demonstrate the practicality of cuZK.

4.1 Setup

We perform the experiments on three testbeds: 1) G3060, 2) V100, and 3) 3700X, whose hardware configurations are shown in Table 1. The testbed V100 is equipped with eight Nvidia Tesla V100 GPU cards. All GPU cards on the testbed V100 are connected with efficient Nvidia NVLink. Experiments on multi-GPU systems are executed on the testbed V100. The testbed G3060 is only equipped with one Nvidia GeForce GTX 3060 GPU card, and its CPU-GPU data transfer is completed through PCI-E. The testbed 3700X is not equipped with any GPU card. It is only used to evaluate the performance of CPU implementations.

Table 2 gives the baseline implementations that we compare in this paper. They are all state-of-the-art zkSNARK implementations, including a single-GPU implementation Mina [19], a multi-GPU implementation Bellperson [20], and two CPU implementations, Libsnark [35] and Bellman [21]. Note that the difference in hardware resources can significantly affect comparison results between CPU and GPU implementations. Therefore, in order to make comparisons in a relatively fair manner, we choose to perform cuZK and other CPU implementations in the two testbeds with chips at a similar price, namely G3060 and 3700X.

TABLE 1: Hardware Configuration of Testbeds

Testbeds	G3060	V100	3700X
Device (GPU)	Nvidia GeForce GTX 3060	Nvidia Tesla V100	/
Core Count	3584	8×5120	/
Core Freq.	1.32 GHz	1.24 GHz	/
Mem Capacity	12 GB	32 GB	/
Mem BandW.	360 GB/sec	900 GB/sec	/
Host (CPU)	AMD Ryzen 3700X	Intel(R) Xeon(R) Platinum 8260	AMD Ryzen 3700X
CPU Cores	8	2×24	8
CPU Freq.	3.60 GHz	2.40 GHz	3.60 GHz
Mem Capacity	32 GB	256 GB	32 GB
OS	Ubuntu 20.04	CentOS 7.8	Ubuntu 20.04

TABLE 2: Some GPU/CPU Implementations of zkSNARK

Implementations	Platforms	Multiple GPUs Supported	Optional Elliptic Curves
cuZK (ours)	GPU	✓	BLS381, MNT4753, ALT-BN128
Bellperson [20]	GPU	✓	BLS381
Mina [19]	GPU	×	MNT4753
Bellman [21]	CPU	/	BLS381
Libsnark [35]	CPU	/	BLS381, MNT4753, ALT-BN128

4.2 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 first evaluate our MSM implementation on single GPU systems. Table 3 gives the evaluation results, including the execution times and speedups over other MSM implementations. We perform these implementations with different elliptic curves due to the limitation of their optional elliptic curves. We choose the curve MNT4753 for Mina, BLS381 for Bellperson and Bellman, and ALT-BN128 for

TABLE 3: Execution Time (sec) and Speedup for MSM Implementations with Different MSM Scales on Single-PU Systems

Size	MNT4753		BLS381				ALT-BN128	
	Mina (V100)	cuZK (V100)	Bellperson (V100)	cuZK (V100)	Bellman (3700X)	cuZK (G3060)	Libsnark (3700X)	cuZK (G3060)
2^{19}	8.701	0.732 (11.89X)	0.241	0.116 (2.08X)	1.235	0.133 (9.29X)	0.767	0.087 (8.82X)
2^{20}	16.071	1.163 (13.82X)	0.409	0.188 (2.18X)	2.391	0.236 (10.13X)	1.468	0.153 (9.59X)
2^{21}	31.789	1.960 (16.22X)	0.727	0.331 (2.20X)	4.795	0.419 (11.44X)	2.763	0.282 (9.80X)
2^{22}	62.344	3.608 (17.28X)	1.301	0.578 (2.25X)	6.375	0.759 (8.40X)	5.259	0.532 (9.89X)
2^{23}	124.429	6.635 (18.75X)	2.637	1.154 (2.29X)	12.559	1.462 (8.59X)	9.999	1.059 (9.44X)

Libsnark. Our implementation supports all three curves. To conclude, we achieve a speedup of up to $11.44\times$ and $9.89\times$ over the CPU implementations addressed in Bellperson and Libsnark, respectively. We achieve a speedup of up to $18.75\times$ and $2.29\times$ over the GPU implementations addressed in Mina and Bellperson, respectively. The performance of the MSM implementation addressed in Mina is relatively terrible because it employs a Straus-based parallel MSM algorithm [48], which cannot perform as well as Pippenger-based algorithms when the scale of MSM is very large.

We also evaluate our MSM implementation on multi-GPU systems. Figure 10 gives their execution times on systems with different numbers of GPUs. Here we only compare with Bellperson because it is the only baseline implementation that supports multi-GPU execution. Our MSM implementation yields up to $2.51\times$ (2GPUs), $2.39\times$ (4GPUs), $2.63\times$ (8GPUs) speedup over that in Bellperson. In addition, we evaluate our MSM implementation 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 11, the throughput of our MSM implementation grows almost linearly with the number of threads until the thread number exceeds GPU core number. Note that we perform the experiment in the testbed V100, where each GPU card has 5120 cores.

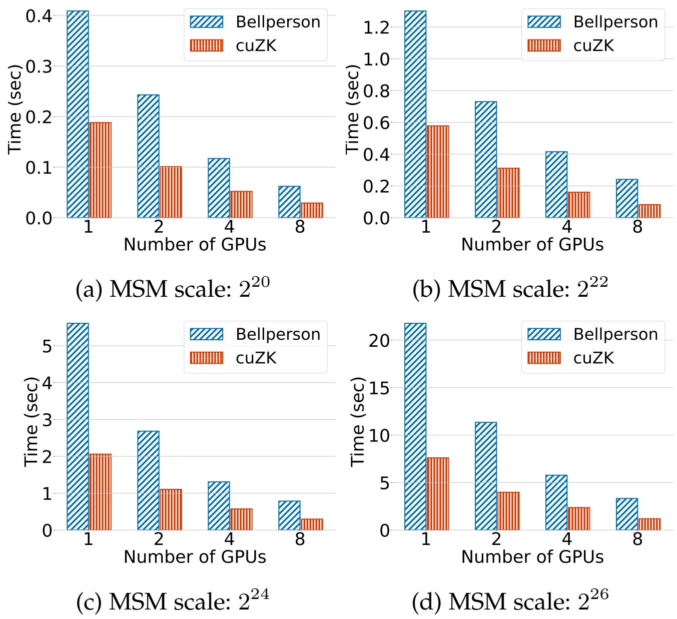


Fig. 10: Execution time for MSM implementations with different MSM scales on single- and multi-GPU systems.

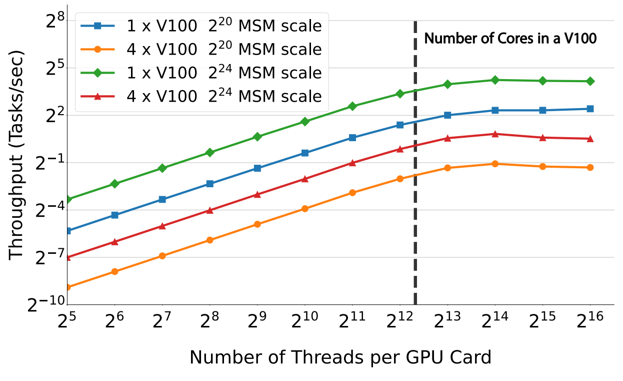


Fig. 11: Throughput for our MSM implementation with the different number of execution threads.

4.3 Evaluating the Overall Performance of cuZK

In this section, we give the overall performance of cuZK. Here, we evaluate the baseline implementation and cuZK using the BLS381 curve and perform all experiments on the testbed V100 with single or multiple GPU cards.

Figure 12 gives the execution times of four zkSNARK operations, namely DT, MUL, NTT, and MSM. The experimental results show that cuZK provides a speedup of up to $16.06\times$, $202.26\times$, and $2.55\times$ over Bellperson for DT, MUL, and MSM, respectively. It also yields $1.70\times$ speedup for NTT on the single-GPU system. Figure 13 gives more intuitive evaluation results. It is obvious that our superiority over the baseline implementation becomes more apparent as the number of GPU cards increases. The overall performance of cuZK achieves over $2.65\times$ (1GPU), $3.02\times$ (2GPU), $3.53\times$ (4GPU), $4.47\times$ (8GPU) speedup.

Below we give a deeper insight into our experimental results. First, as shown in Figure 13, our MSM part takes most of the overall runtime of cuZK, while the other parts take only a little bit in total. This is actually the result of our huge improvements in the DT and MUL implementations. Second, as shown in Figure 12(b)(c), the lines corresponding to the execution times of our MUL and NTT implementations on single- and multi-GPU systems are nearly coincident. This is because we perform all NTTs and MULs in each GPU card instead of splitting them across GPUs. Bellperson uses

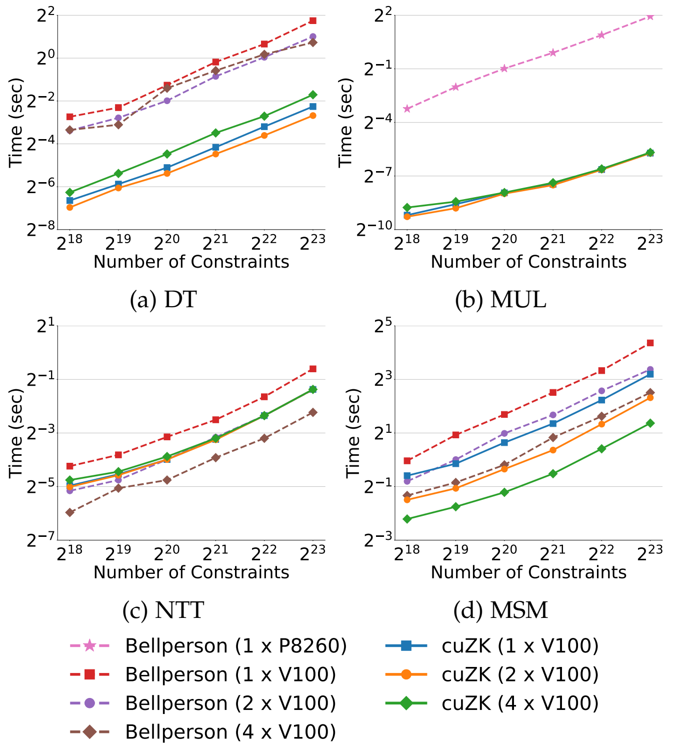


Fig. 12: The execution times of four zkSNARK operations, namely DT, MUL, NTT, and MSM. P8260 represents the CPU used in the testbed V100.

at most three GPUs (if available) to execute independent NTTs simultaneously. However, as a trade-off, it introduces additional overheads on GPU-GPU data transfer, leading to its overall performance gain being limited.

4.4 Evaluating cuZK in Real-world Applications

Finally, we evaluate cuZK in real-world applications to demonstrate its practicality. We choose to employ cuZK in two cryptocurrency applications, namely ZCash (ZC) [9] and Filecoin (FL) [10], and three classical cryptographic workloads [49], namely Merkle Tree (MT), Hybrid Encryption (HE), and Augmented Auction (AA). We compare these updated implementations with their original CPU/GPU implementations. The comparison results are shown in Figure 14. Note that there is unfortunately no existing GPU implementation for the other four applications except Filecoin, and thus we can only evaluate them with their CPU implementations. As a result, cuZK provides a speedup of 2.18× over the original GPU implementation of Filecoin, and a speedup of up to 8.83× when compared with their CPU implementations.

5 Conclusion

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. Actually, 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. As a result, our evaluation shows cuZK has a considerable speedup over other state-of-the-art GPU implementations of zkSNARK.

Furthermore, our work can be extended to other ZKP protocols that also require these zkSNAKR operations, namely MUL, MSM, and NTT. In the future, we plan to explore more GPU-accelerated methods for a wider range of ZKP protocols.

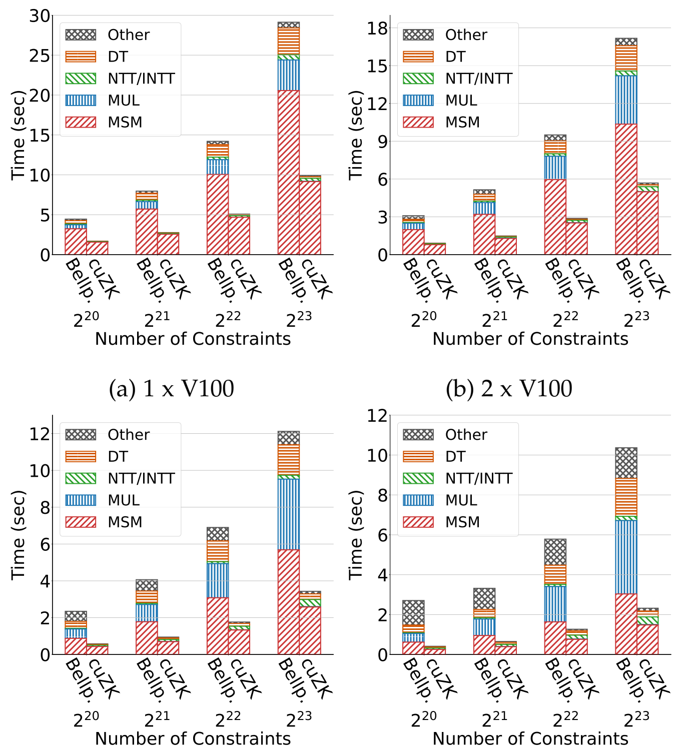


Fig. 13: The overall execution time of cuZK and Bellperson with the different number of constraints in the R1CS instance.

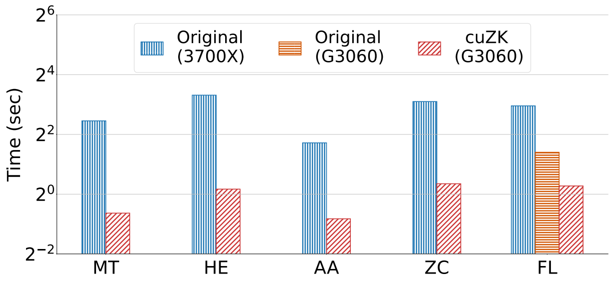


Fig. 14: The execution time of cuZK in real-world applications.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[2] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.

[3] J. Kilian, "A note on efficient zero-knowledge proofs and arguments," in *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, 1992, pp. 723–732.

[4] S. Micali, "Computationally sound proofs," *SIAM Journal on Computing*, vol. 30, no. 4, pp. 1253–1298, 2000.

[5] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2010, pp. 321–340.

[6] Z. Zhao and T.-H. H. Chan, "How to vote privately using bitcoin," in *International Conference on Information and Communications Security*. Springer, 2015, pp. 82–96.

[7] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vsql: Verifying arbitrary sql queries over dynamic outsourced databases," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 863–880.

[8] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 397–411.

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

[10] J. Benet and N. Greco, "Filecoin: A decentralized storage network," *Protocol Labs*, pp. 1–36, 2017.

[11] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Coda: Decentralized cryptocurrency at scale," *Cryptology ePrint Archive*, 2020.

[12] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng, "Veriml: Enabling integrity assurances and fair payments for machine learning as a service," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2524–2540, 2021.

[13] J. Zhang, Z. Fang, Y. Zhang, and D. Song, "Zero knowledge proofs for decision tree predictions and accuracy," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2039–2053.

[14] G. Lu, W. Zhang, and Z. Wang, "Optimizing depthwise separable convolution operations on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 70–87, 2021.

[15] "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," 2012. [Online]. Available: https://github.com/pytorch/pytorch

[16] Y. Gao, J. Xu, and H. Wang, "Cunh: Efficient gpu implementations of post-quantum kem newhope," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 551–568, 2021.

[17] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. K. M. Mi, "Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 379–391, 2020.

[18] X. Wang, Y. Qiu, S. R. Slattery, Y. Fang, M. Li, S.-C. Zhu, Y. Zhu, M. Tang, D. Manocha, and C. Jiang, "A massively parallel and scalable multi-gpu material point method," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 4, pp. 30–1, 2020.

[19] "Gpu groth16 prover," 2019. [Online]. Available: https://github.com/MinaProtocol/gpu-groth16-prover-3x

[20] "bellperson: Gpu parallel acceleration for zksnark," 2019. [Online]. Available: https://github.com/filecoin-project/bellperson

[21] "bellman: a crate for building zksnark circuits," 2015. [Online]. Available: https://github.com/zkcrypto/bellman

[22] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "Pqc acceleration using gpus: Frodokem, newhope, and kyber," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, 2020.

[23] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 264–275.

[24] "Opencl: Open standard for parallel programming of heterogeneous systems," 2009. [Online]. Available: https://www.khronos.org/opencl

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

[26] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[27] J. L. Greathouse and M. 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*. IEEE, 2014, pp. 769–780.

[28] Y. Tao, Y. Deng, S. Mu, M. Zhu, L. Xiao, L. Ruan, and Z. Huang, "Atomic reduction based sparse matrix-transpose vector multiplication on gpus," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 987–992.

[29] J. H. Silverman, *The arithmetic of elliptic curves*. Springer, 2009, vol. 106.

[30] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[31] P. d. Rooij, "Efficient exponentiation using precomputation and vector addition chains," in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1994, pp. 389–399.

[32] C.-C. Chang and D.-C. 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*. IEEE, 2003, pp. 955–958.

[33] D. J. Bernstein, J. Doumen, T. Lange, and J.-J. Oosterwijk, "Faster batch forgery identification," in *International Conference on Cryptology in India*. Springer, 2012, pp. 454–473.

[34] "gnark-crypto: gnark-crypto provides efficient cryptographic primitives in go." 2020. [Online]. Available: https://github.com/ConsenSys/gnark-crypto.git

[35] "libsnark: a c++ library for zksnark proofs," 2014. [Online]. Available: https://github.com/scipr-lab/libsnark

[36] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "Dizk: A distributed zero knowledge proof system," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 675–692.

[37] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[38] D. Kincaid, T. Oppe, and D. Young, "Itpackv 2d user's guide," 1989. [Online]. Available: https://web.ma.utexas.edu/CNA/ITPACK/manuals/userv2d

[39] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[40] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: A c++ templated sparse matrix library," *URL http://cusplibrary. github. io. Accessed: December*, 2014.

[41] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *International Workshop on Applied Parallel Computing*. Springer, 2006, pp. 260–269.

[42] E.-J. Im and K. Yelick, "Optimization of sparse matrix kernels for data mining," in *submitted to First SIAM Conf. on Data Mining*. Citeseer, 2000.

[43] C. Chen, J. Zhou, L. Wang, X. Wu, W. Fang, J. Tan, L. Wang, A. X. Liu, H. Wang, and C. Hong, "When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 2652–2662.

[44] "Cub: Cooperative primitives for cuda c++." 2013. [Online]. Available: https://nvlabs.github.io/cub

[45] M. Garland, "Sparse matrix computations on manycore gpu's," in *Proceedings of the 45th annual design automation conference*, 2008, pp. 2–6.

[46] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

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

[48] E. G. Straus, "Addition chains of vectors (problem 5125)," *American Mathematical Monthly*, vol. 70, no. 806-808, p. 16, 1964.

[49] "jsnark: A java library for zk-snark circuits," 2015. [Online]. Available: https://github.com/akosba/jsnark