Elastic MSM: A Fast, Elastic and Modular Preprocessing Technique for Multi-Scalar Multiplication Algorithm on GPUs

Xudong Zhu[12], Haoqi He[12], Zhengbang Yang[12], Yi Deng[12], Lutan Zhao[12] and Rui Hou[12]

[1] Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China,

[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China,

{zhuxudong,hehaoqi,yangzhengbang,deng,zhaolutan,hourui}@iie.ac.cn

Abstract. Zero-knowledge proof (ZKP) is a cryptographic primitive that enable a prover to convince a verifier that a statement is true without revealing any other information beyond the correctness of the statement itself. Due to its powerful capabilities, its most practical type, called zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK), has been widely deployed in various privacy-preserving applications such as cryptocurrencies and verifiable computation. Although state-of-the-art zkSNARKs are highly efficient for the verifier, the computational overhead for the prover is still orders of magnitude too high to warrant use in many applications. This overhead is due to several time-consuming operations, including large-scale matrix-vector multiplication (MUL), number-theoretic transform (NTT), and especially the multi-scalar multiplication (MSM) with the highest proportion. Thus, further efficiency improvements are needed.

In this paper we focus on comprehensive optimization of running time and storage space needed by the MSM algorithm on GPUs. Specifically, we propose a new modular and adaptive parameter configuration technique—*elastic MSM* to enable us to change the scale of MSM according to our own wishes by performing a corresponding amount of preprocessing. This technique enable us to fully unleash the potential of various efficient parallel MSM algorithms. From another perspective, our technique could also be regarded as a preprocessing technique over the well-known Pippenger algorithm, which is modular and could be used to accelerate almost all the most advanced parallel Pippenger algorithms on GPUs. Meanwhile, our technique provides an adaptive trade-off between the running time and the extra storage space needed by parallel Pippenger algorithms on GPUs. We implemented and tested *elastic MSM* over two prevailing parallel Pippenger algorithms on GPUs. Given a range of practical parameters, across various preprocessing space limitations (across various MSM scales), our constructions achieve up to about 28× and 45× (25× and 40×) speedup versus two state-of-the-art preprocessing parallel Pippenger algorithms on GPUs, respectively.

Keywords: Zero-Knowledge Proof · Multi-Scalar Multiplication (MSM) · Parallel Algorithm · Graphics Processing Unit (GPU)

1 Introduction

The proposal of a zero-knowledge argument system [GMR89], especially the *Non-Interactive Zero-Knowledge argument system* (NIZK) [BFM88], has a significant impact on both cryptography theory research and the application of cryptography. In the last decade, remarkable progress has been made in research on *zero-knowledge Succinct Non-interactive ARgument of Knowledge* (zkSNARK). After various studies [Kil92, Mic00, Gro10, Lip12, GGPR13], Groth constructed a pairing-based zkSNARK [Gro16] with only hundreds of bytes proof and is very fast to be verified within several milliseconds. Then, zkSNARKs are widely considered to be the most practical zero-knowledge proofs, and they have been widely applied to privacy-preserving applications, such as verifiable database outsourcing [ZGK⁺17], verifiable machine learning [ZFZS20], privacy-preserving cryptocurrencies [BSCG⁺14, BG17, BMRS20], electronic voting [ZC16], online auction [GY19], and anonymous credentials [DLFKP16].

Although proofs of the most state-of-the-art zkSNARKs such as [Gro16, BBB⁺18, GWC19, CHM⁺20] are succinct and fast to verify, their generation remains a bottleneck in large-scale zkSNARK adoption. For example, to generate a proof for a program (which could be translated into a usually several times larger constraint system), the prover in these zkSNARKs need to perform various time-consuming operations, such as large-scale matrix-vector multiplication (MUL), number-theoretic transform (NTT), and the multi-scalar multiplication (MSM) on elliptic curves. And the number of operations required is always super-linear comparing to the number of constraints. As a consequence, it takes much longer to generate the zkSNARK proof of a program, and could be up to a few minutes just for a single payment transaction [BSCG⁺14]. Notably, among these expensive operations, [Xav22, LWY⁺23] have mentioned that for some of the most popular zkSNARK protocols, MSM is the most time-consuming operation, taking more than 70 percent of the total runtime.

An important research line involves reducing the proof generation time by specifically designing MSM operations on certain hardware, including GPUs [Min19, Bel19, Spp22, Yrr22, Mat22, MXS⁺23, LWY⁺23], FPGAs [Xav22, ABC⁺22, Har22], ASICs [ZWZ⁺21], and CPU clusters [WZC⁺18]. Although DIZK [WZC⁺18] could accelerate the proof generation algorithm by distributing this algorithm to CPU clusters, it is still not suitable for widespread deployment due to much higher deployment overhead for CPU clusters than GPU cards, FPGA chips and ASIC chips. And due to extremely high development costs of ASIC, the ASIC design [ZWZ⁺21] is not yet widely adopted. Therefore, most recent research focus on improving the efficiency of MSM on GPUs or FPGAs.

1.1 Our Contributions

In this paper, our motivation is to design a new adaptive parameter configuration module that is compatible with almost all the existing most efficient MSM algorithms on GPUs, and therefore further speed-up all these algorithms. Contributions of this paper are summarized below:

- **Propose a new modular and adaptive parameter configuration technique**. With this series of research [Min19, Bel19, Spp22, Yrr22, Mat22, LWY⁺23], the fact that more and more potential of GPUs has been unleashed leads to better and better performance on the MSM parallel computations. Especially, the faster parallel MSM algorithm proposed in [LWY⁺23] has achieved nearly perfect linear speedup over the Pippenger algorithm [Pip76]. Thus, it seems difficult to further accelerate the parallel MSM algorithm from the MSM computation algorithm itself. So we change our perspective and try to find a new technique *elastic MSM* which enable us to adaptively change the scale of MSM (with sacrifice in storage space). Notably, while different algorithms determine different computational complexities, our technique which is independent and compatible with algorithmic improvements endows us with the freedom to adjust parameters to maximize the power of a particular algorithm.
- **Analysis the advantages of *elastic MSM* in theory**. Interestingly, we find that the *Computation Consolidation* technique proposed in [MXS⁺23] could be

regarded as a special case of *elastic MSM*, that is *elastic MSM* with parameter fixed to a specific value. Therefore, our theory could provide a theoretical support for their technique from a higher dimension. With our plug-and-play technique *elastic MSM*, for the same MSM instance, regardless of which Pippenger like algorithm is used, we can freely adjust the instance parameters according to the different parameter advantage intervals of each algorithm to fully unleash the potential of the algorithm. As an example, we combine our *elastic MSM* with two parallel MSM algorithms, to obtain two efficient preprocessing parallel Pippenger algorithms—*elastic Pippengers*, and theoretically analyze their advantages over the preprocessing parallel Pippenger algorithms obtained by combing these two parallel MSM algorithms with the preprocessing technique proposed in [MXS+23]. More specifically, the *elastic Pippengers* have better time-space flexibility. With the same preprocessing space, our *elastic Pippengers* require less PDBLs during the preprocessing. Moreover, our *elastic Pippengers* theoretically achieve significant speedup on the MSM PADDs while only increasing a little extra MSM PDBLs overhead.

- **Design some evaluation schemes for *elastic MSM*.** We plug both our preprocessing techniques and the preprocessing techniques proposed in GZKP [MXS+23] in two common parallel Pippenger algorithms on GPUs to obtain four state-of-the-art preprocessing parallel Pippenger implementations on GPUs. When we set preprocessing storage space limitations to be $7 \cdot 2^{22}, 5 \cdot 2^{22}, 3 \cdot 2^{22}, 2 \cdot 2^{22}$ and 2^{22} extra EC points, the evaluation results show that with other practical parameters fixed, our MSM algorithm delivers a speedup over 4.0× and up to 28.2×, and 11.8× and up to 45.2× in two implementations, respectively. Additionally, across MSM scales from 2^{16} to 2^{22} and with other practical parameters fixed, our MSM algorithm delivers a speedup over 10.5× and up to 25.5×, and 14.4× and up to 40.8× in two implementations, respectively. The details of the evaluations are described in **Table 5**, **Table 6**, **Table 7** and **Table 8**.

2 Preliminaries

2.1 Elliptic curve

An elliptic curve (EC) 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 $\mathcal{O}$, could be served as the identity element in the abelian group formed by all EC points.

These points support several common operations, including point addition (PADD), point doubling (PDBL) and point scalar multiplication (PMULT), where PADD is the fundamental operation. PDBL is a special case of PADD, the result of which is equal to performing a PADD operation on two identical points. As for PMULT of a scalar k and an EC point $\mathbf{P}$, it can be defined as k times self-PADD of $\mathbf{P}$, denoted by $k\mathbf{P} = \mathbf{P} + \mathbf{P} + ... + \mathbf{P}$. In fact, the pair $k\mathbf{P}$ is always computed by using double-and-add method to execute a series of PDBLs and PADDs, we show an example of computing $25\mathbf{P}$ in **Figure 1** to explain this method. We can first represent 25 in its binary form $(11001)_2$ and initialize the result to be the point at infinity $\mathcal{O}$. Then at each bit position, we execute a PDBL to double the point. If the bit is 1, we add it to the result using a PADD.

2.2 Multi-scalar Multiplication

MSM performs the vector inner-product on the exponents of group elements. Specifically, The $(n, \lambda)-$MSM is defined as follows: given a vector of group elements $(\mathbf{P}_1, \dots, \mathbf{P}_n)$, we need to compute the formula $Q = \sum_{i=1}^{n} k_i \mathbf{P}_i$, using the minimal number of multiplications

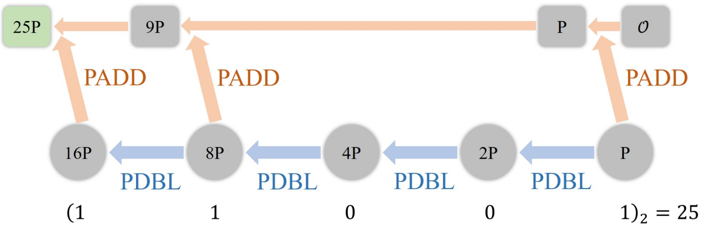


Figure 1: An example of PMULT computation. $\mathcal{O}$ is the point at infinity on elliptic curve.

possible, where $\{k_i\}_{i\in[1,n]}$ are all λ bits scalar. If we compute MSM by performing PMULT for each $k_i\mathbf{P}_i$ and then add all pairs $k_i\mathbf{P}_i$ directly, it is clear that there are many expensive PMULT operations on an EC are needed. Based on our discussion in **Subsection 2.1**, if we simply 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. However in real-world applications, the security parameter λ commonly ranges from 254 to 768 and the scale of MSM n could be larger than a million and up to several billion. Even worse, the costs of EC point operations like PADD and PDBL themselves are much more expensive than the regular scalar operations. Altogether, the computational cost of simply using the double-and-add method for MSM calculation is not acceptable. Therefore, some research on more efficient MSM algorithms has emerged, such as the Pippenger algorithm [Pip76], the Chang-Lou algorithm [CL03], and the Bos-Coster algorithm reported in [dR95]. Especially, the Pippenger algorithm performs best when the scale of MSM is very large, as shown in [BDLO12].

2.3 MSM in zkSNARK

zkSNARK is an application oriented variant of ZKP, which is one of the most important and widely used primitives in cryptography. A zkSNARK could be used by the prover to convince the verifier that, "given a public relation R and a pubic statement x, I know a secret witness w such that $R(x, w) = 1$", without leaking any other information. While there are several different definitions about zkSNARK, we will provide a brief introduction to the one known as *publicly-verifiable preprocessing zkSNARK* (see [BCI$^+$13, GGPR13] for details). Informally, zkSNARK consists of three algorithms (Setup, Prove, Verify) (see **Figure 2**) such that

- $(pk_R, vk_R) \leftarrow \text{Setup}(R)$: On input the relation R, this probabilistic algorithm output a proving key pk_R and the verification key vk_R. Both keys are public parameters and they could be used to prove/verify any number of statements about relation R. That is, the Setup needs to be run only once for a specific relation R.

- $\pi \leftarrow \text{Prove}(R, pk_R, x, w)$: For the public relation R, The probabilistic prover receives the proving key pk_R, a public input x for R and a secret input w for R, outputs a proof π to prove that "I know a secret w such that $R(x, w) = 1$". Notably, the generation of π involves randomness that ensures its zero knowledge property.

- $0/1 \leftarrow \text{Verify}(R, vk_R, x, \pi)$: For the public relation R, The deterministic verifier receives the verification key vk_R, a public input x for R and a proof π, outputs a decision bit 0/1 ("accept" or "reject").

zkSNARK should satisfy the following five properties: (1)*completeness*, i.e., honestly generated proof can be verified. (2)*knowledge soundness*, i.e., if the proof is verified by the verifier, then the prover must know the witness w such that $R(x, w) = 1$. (3)*zero knowledge*, i.e., there is no any other information could be derived from the proof other than that the statement x is true. (4)*succinctness*, i.e., small proof sizes and fast verification regardless of R's complexity. (5)*non-interactive*, i.e., only a single message from the prover to the verifier.

It can be seen from the definition that the design intention of zkSNARK determines that itself has a small proof size and a fast verification. This allows zkSNARKs to serve as a foundation for highly efficient verifiable computation. However, The trade-off in the prover's complexity becomes a computational bottleneck. If we take apart the computation of the prover, just as shown in the data of [Xav22, LWY+23], the MSM operation accounts for the vast majority of the proportion (more than 70% for many state-of-the-art zkSNARKs), making it the main computational bottleneck of the zkSNARK prover's complexity.

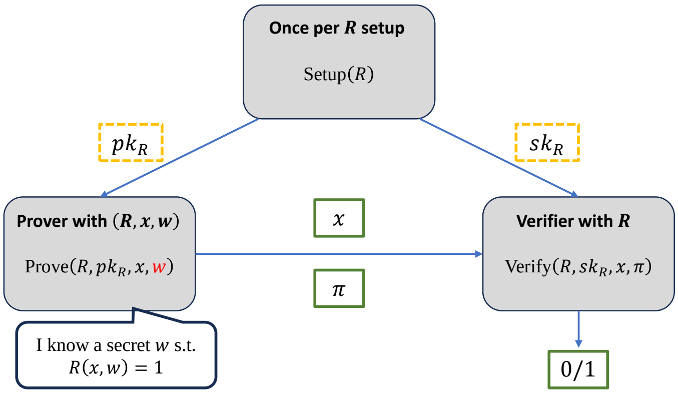


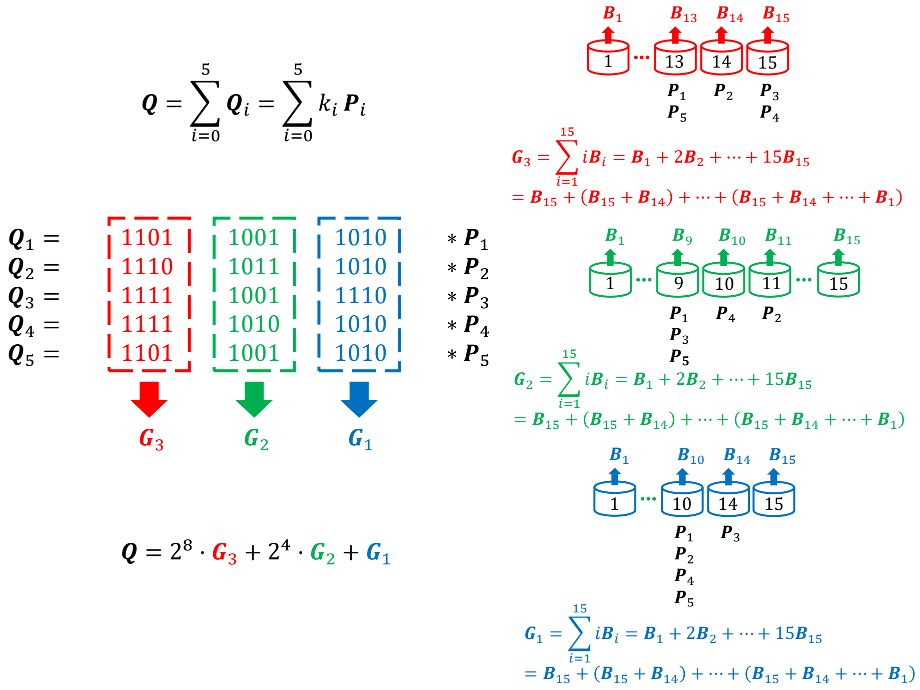
Figure 2: Components of a zkSNARK.

2.4 The Pippenger Algorithm

Four decades ago, Pippenger provided an asymptotically optimal algorithm for the MSM in [Pip76]. To this day, the Pippenger algorithm and its variants are still the state-of-the-art and widely used algorithms. It is noticeable that our *elastic MSM* technique could be compatible with all the Pippenger-like parallel MSM algorithms. And we review the Pippenger algorithm and analyze its computational costs in this subsection (also see a simple example in **Figure 3**).

Algorithm Description. If we set the scale of MSM to be n, the bit length of scalar to be λ and the window size to be s (without losing generality, we assume that λ is divisible by s in our paper), to compute the MSM $\mathbf{Q} = \sum_{i=1}^{n} \mathbf{Q}_i = \sum_{i=1}^{n} k_i \mathbf{P}_i$, then the Pippenger algorithm consists of these three steps:

1. Convert the task using *windowed MSM* technique. We first divide the original task $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i$ into multiple smaller subtasks according to the window, and we denote this step by *windowed MSM*. With the window size s, it is clear that we could divide each λ bits scalar k_i into $\frac{\lambda}{s}$ parts, and each part is a s bits scalar m_{ij}, satisfying $k_i = \sum_{j=1}^{\frac{\lambda}{s}} 2^{(j-1)s} m_{ij}$. Then the smaller subtasks are defined as the computation $\mathbf{G}_j = \sum_{i=1}^{n} m_{ij} \mathbf{P}_i$, where $j \in \left[1, \frac{\lambda}{s}\right]$. Therefore, the relation between the original task and these subtasks can be expressed by **Formula (1)**



$Q = \sum_{i=0}^{5} Q_i = \sum_{i=0}^{5} k_i P_i$
$Q_1 =$ 1101 1001 1010 $* P_1$
$Q_2 =$ 1110 1011 1010 $* P_2$
$Q_3 =$ 1111 1001 1110 $* P_3$
$Q_4 =$ 1111 1010 1010 $* P_4$
$Q_5 =$ 1101 1001 1010 $* P_5$
G_3
G_2
G_1
$Q = 2^8 \cdot G_3 + 2^4 \cdot G_2 + G_1$
B_1 B_{13} B_{14} B_{15}
1 13 14 15
P_1 P_5 P_2 P_3 P_4
$G_3 = \sum_{i=1}^{15} iB_i = B_1 + 2B_2 + \cdots + 15B_{15}$
$= B_{15} + (B_{15} + B_{14}) + \cdots + (B_{15} + B_{14} + \cdots + B_1)$
B_1 B_9 B_{10} B_{11} B_{15}
1 9 10 11 15
P_1 P_3 P_5 P_4 P_2
$G_2 = \sum_{i=1}^{15} iB_i = B_1 + 2B_2 + \cdots + 15B_{15}$
$= B_{15} + (B_{15} + B_{14}) + \cdots + (B_{15} + B_{14} + \cdots + B_1)$
B_1 B_{10} B_{14} B_{15}
1 10 14 15
P_1 P_2 P_4 P_5 P_3
$G_1 = \sum_{i=1}^{15} iB_i = B_1 + 2B_2 + \cdots + 15B_{15}$
$= B_{15} + (B_{15} + B_{14}) + \cdots + (B_{15} + B_{14} + \cdots + B_1)$

Figure 3: An example of Pippenger algorithm with 5 MSM scale, 12-bits scalars, and window size 4.

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

2. **Compute subtask results $\mathbf{G}_j$, for each $j \in \left[1, \frac{\lambda}{s}\right]$.** In each subtask (indexed by j), we put EC points $\mathbf{P}_i$ with the same scalar value m_{ij} into the specific bucket whose index is equal to m_{ij}. Notably, the points corresponding to zero scalars have no effect on the final result, so there are only $2^s - 1$ buckets needed. Then to compute the subtask results $\mathbf{G}_j$, we only need to perform the following two steps:

   (a) Sum all points in the bucket to obtain $\mathbf{B}_l^{(j)}$, for each $l \in [1, 2^s - 1]$. It is obvious that in each subtask, the sum of all points in the buckets requires at most n PADDs.

   (b) Sum all bucket points weighted by their bucket indexes, namely $\mathbf{G}_j = \sum_{l=1}^{2^s-1} l\mathbf{B}_l^{(j)}$, to obtain $\mathbf{G}_j$. Obviously, $\sum_{l=1}^{2^s-1} l\mathbf{B}_l^{(j)} = \sum_{i=1}^{n} m_{ij}\mathbf{P}_i = \mathbf{G}_j$. And by using an efficient algorithm proposed in [BDLO12] (as shown in **Algorithm 1**), the computation of subtask result $\mathbf{G}_j$ requires at most $2^{s+1} - 2$ PADDs.

3. **Compute the MSM result with subtask results, namely $\mathbf{Q} = \sum_{j=1}^{\frac{\lambda}{s}} 2^{(j-1)s}\mathbf{G}_j$.**

Note that:

$$\begin{aligned}\mathbf{Q} &= \sum_{j=1}^{\frac{\lambda}{s}} 2^{(j-1)s}\mathbf{G}_j \\ &= 2^s\left(\cdots\left(2^s\left(2^s\mathbf{G}_{\frac{\lambda}{s}} + \mathbf{G}_{\frac{\lambda}{s}-1}\right) + \mathbf{G}_{\frac{\lambda}{s}-2}\right)\cdots\right) + \mathbf{G}_1\end{aligned} \tag{2}$$

So, computing the MSM result requires at most $\left(\frac{\lambda}{s} - 1\right) s$ PDBLs and $\frac{\lambda}{s}$ PADDs. Then this computation requires at most $\lambda - s$ PDBLs and $\frac{\lambda}{s}$ PADDs.

In summary, 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 1**. And to add the subtask results to the final result, a recursive method based on **Formula (2)** can be used, which requires around $\lambda - s$ PDBLs and $\frac{\lambda}{s}$ PADDs. Since there are $\frac{\lambda}{s}$ subtasks, the total computational consts of the Pippenger algorithm are around $\frac{\lambda}{s}\left(n + 2^{s+1}\right)$ PADDs plus $\lambda - s$ PDBLs. Note that the value of $\frac{\lambda}{s}$ and s are usually small, so it is common for us to omit them in the addition or subtraction terms. However, we explicitly express s in the complexity analysis of PDBLs because it will affect our discussion in **Subsection 3.2**. Moreover, we skip the costs of scalar operations here because they are negligible compared to the costs of EC point operations.

Algorithm 1 BucketPointsReduction [BDLO12]

```
Require: A point vector $\overrightarrow{\mathbf{B}}^{(j)}_{2^s-1} = \left[\mathbf{B}^{(j)}_1, \mathbf{B}^{(j)}_2, \ldots, \mathbf{B}^{(j)}_{2^s-1}\right]$
Ensure: $\mathbf{G}_j = \sum_{l=1}^{2^s-1} l\mathbf{B}^{(j)}_l$
  $\mathbf{G}_{j,0} \leftarrow \mathcal{O}$; $\mathbf{M}_0 \leftarrow \mathcal{O}$    // $\mathcal{O}$ is the point at infinity on the elliptic curve.
  for $l \leftarrow 1$ to $2^s - 1$ do    // Add $l\mathbf{B}^{(j)}_l$ as
      $\mathbf{M}_l \leftarrow \mathbf{M}_{l-1} + \mathbf{B}^{(j)}_{2^s-l}$    // $\mathbf{M}_l = \mathbf{B}^{(j)}_{2^s-1} + \mathbf{B}^{(j)}_{2^s-2} + \cdots + \mathbf{B}^{(j)}_{2^s-l}$
      $\mathbf{G}_{j,l} \leftarrow \mathbf{G}_{j,l-1} + \mathbf{M}_l$    // $\mathbf{G}_{j,l} = \mathbf{M}_1 + \mathbf{M}_2 + \cdots + \mathbf{M}_l$
  end for
  $\mathbf{G}_j \leftarrow \mathbf{G}_{j,2^s-1}$
  return $\mathbf{G}_j$
```

2.5 Graphics Processing Units

Graphics Processing Units (GPUs) are platforms composed of hundreds or even thousands of cores that can handle thousands of threads simultaneously. This makes them particularly well-suited for tasks that can be broken down into many smaller tasks that can be executed in parallel, such as matrix operations, convolutional neural networks, and physical simulations. 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.

GPUs has a special execution model—Single Instruction, Multiple Threads (SIMT) execution model that executes batches of threads in lockstep. In the SIMT model, threads executing the same instruction are grouped into a fixed-sized batch, called a wavefront (AMD) or a warp (NVIDIA). The threads of a batch always execute the same instruction in lockstep on a single instruction, multiple data (SIMD) unit, i.e., in parallel on different operands.

3 A New Parameter Configuration Technique

Recently, there are many excellent works such as [Min19, Bel19, Spp22, Yrr22, Mat22, LWY$^+$23] trying to optimize the parallel Pippenger algorithm itself. Especially, the faster parallel Pippenger-like algorithm proposed in [LWY$^+$23] has achieved nearly perfect linear speedup over the Pippenger algorithm [Pip76], where perfect linear speedup means the parallel speedup ratio is equal to the number of execution threads. Although this road is thriving, we find that there is another parallel shortcut to efficiency that is often overlooked.

In our opinion, the optimization of parallel MSM algorithms could be divided into two independent and mutually beneficial directions. While one direction focuses on MSM algorithm optimization itself, which is currently a popular direction. The other direction concentrates on techniques to optimize and adjust the scale of MSM to better adapt to a certain MSM algorithm, which is exactly what we will delve into in this papper. In fact, as shown in [BDLO12], the Pippenger algorithm performs best when the scale of MSM is very large. Also as noticed in [LWY$^+$23], the advantage of Pippenger algorithm only comes when there are a great amount of EC points placed into the same buckets and processed as a whole, which means that the larger the scale of MSMs is, the more benefits this advantage brings.

3.1 The Elastic MSM Algorithm

Recall that the first step of the Pippenger algorithm is to divide the original task into multiple smaller subtasks, using *windowed MSM* technique. In this subsection, we will present a new modular and adaptive parameter configuration technique—*elastic MSM*, which, like *windowed MSM*, could also be regarded as a task partitioning scheme.

From **Formula (1)**, it is obvious that we could deduce the equation $\mathbf{Q} = \sum_{i=1}^{n} \mathbf{Q}_i = \sum_{i=1}^{n} \sum_{j=1}^{\frac{\lambda}{s}} \left(2^{(j-1)s} m_{ij}\right) \mathbf{P}_i$, thus for each $i \in [1, n]$ we have $\mathbf{Q}_i = \sum_{j=1}^{\frac{\lambda}{s}} \left(2^{(j-1)s} m_{ij}\right) \mathbf{P}_i$. Now we define the w, k such that $\frac{\lambda}{s} = w \cdot k$, this equation could be equivalently expressed in the form of a matrix multiplication:

$$\mathbf{Q}_i = \begin{pmatrix} 1 & 2^s & 2^{2s} & \cdots & 2^{(wk-1)s} \end{pmatrix} \cdot \begin{pmatrix} m_{i1}\mathbf{P}_i \\ m_{i2}\mathbf{P}_i \\ m_{i3}\mathbf{P}_i \\ \vdots \\ m_{i(wk)}\mathbf{P}_i \end{pmatrix} \tag{3}$$

With the $\frac{\lambda}{s} = w \cdot k$, our ideas come from an observation that for each $i \in [1, n]$, if we define $M_{i(l,t)} := m_{i((l-1)k+t)}$, where $l \in [1, w]$ and $t \in [1, k]$, then the following equation holds:

$$\mathbf{Q}_i = \sum_{j=1}^{wk} \left(2^{(j-1)s} m_{ij}\right) \mathbf{P}_i = \sum_{l=1}^{w} \sum_{t=1}^{k} 2^{((l-1)k+(t-1))s} M_{i(l,t)} \mathbf{P}_i$$

This means that except the **Formula (3)**, $\mathbf{Q}_i$ could also be equivalently expressed as another form of matrix multiplication:

$$\begin{pmatrix} 1 & 2^{ks} & \cdots & 2^{(w-1)ks} \end{pmatrix} \cdot \mathbf{P}_i \cdot \begin{pmatrix} M_{i(1,1)} & M_{i(1,2)} & \cdots & M_{i(1,k)} \\ M_{i(2,1)} & M_{i(2,2)} & \cdots & M_{i(2,k)} \\ \vdots & \vdots & & \vdots \\ M_{i(w,1)} & M_{i(w,2)} & \cdots & M_{i(w,k)} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2^s \\ \vdots \\ 2^{(k-1)s} \end{pmatrix} \tag{4}$$

If we define $\mathbf{P}_{ij} := 2^{(j-1)ks}\mathbf{P}_i$, $\mathbf{G}_{il} := \sum_{j=1}^{w} M_{i(j,l)}\mathbf{P}_{ij}$ and $N_{ij} := \sum_{l=1}^{k} 2^{(l-1)s}M_{i(j,l)}$, where $i \in [1,n]$ and $j \in [1,w]$, then the **Formula (4)** can be deduced to the **Formula (5)**, which is actually a formula similar to the **Formula (1)**.

$$\begin{aligned}\mathbf{Q}_i = \sum_{j=1}^{w} N_{ij}\mathbf{P}_{ij} &= \sum_{j=1}^{w}\sum_{l=1}^{k} 2^{(l-1)s}M_{i(j,l)}\mathbf{P}_{ij} \\ &= \sum_{l=1}^{k} 2^{(l-1)s}\sum_{j=1}^{w} M_{i(j,l)}\mathbf{P}_{ij} \\ &= \sum_{l=1}^{k} 2^{(l-1)s}\mathbf{G}_{il}\end{aligned} \tag{5}$$

By summarizing the above derivation, we then obtain our core technique—*elastic MSM*. For $\frac{\lambda}{s} = w \cdot k$, $i \in [1,n]$ and $j \in [1,w]$ and $l \in [1,k]$, if we denote $N_l^{((i-1)w+j)} := M_{i(j,l)}$, $\mathbf{P}^{((i-1)w+j)} := \mathbf{P}_{ij}$ and $\mathbf{G}^{(l)} := \sum_{t=1}^{nw} N_l^{(t)}\mathbf{P}^{(t)}$, then the whole computation of MSM can be represented by **Formula (6)**.

$$\begin{aligned}\mathbf{Q} = \sum_{i=1}^{n}\mathbf{Q}_i &= \sum_{i=1}^{n}\sum_{j=1}^{w} N_{ij}\mathbf{P}_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{w}\sum_{l=1}^{k} 2^{(l-1)s}M_{i(j,l)}\mathbf{P}_{ij} \\ &= \sum_{t=1}^{nw}\sum_{l=1}^{k} 2^{(l-1)s}N_l^{(t)}\mathbf{P}^{(t)} = \sum_{l=1}^{k} 2^{(l-1)s}\sum_{t=1}^{nw} N_l^{(t)}\mathbf{P}^{(t)} \\ &= \sum_{l=1}^{k} 2^{(l-1)s}\mathbf{G}^{(l)}\end{aligned} \tag{6}$$

Note that this **Formula (6)** obtained from our *elastic MSM* is very similar to the **Formula (1)** obtained from the trivial *windowed MSM*. So we could also perform the Pippenger algorithm according to the **Formula (6)** rather than **Formula (1)**. Specifically, We divide the coefficients of each $\mathbf{Q}_i$ into windows and express them as a $w \times k$ matrix (each element in this matrix is still a s bit window), instead of expressing them as a long vector just like *windowed MSM*. That is, with $\frac{\lambda}{s} = w \cdot k$, both our *elastic MSM* and *windowed MSM* choose the same window size s. However, we consider the scale of MSM as wn rather than n and the bit length of scalar as ks rather than λ (see **Figure 4** and **Figure 5** for easier understanding). Moreover, the computation in our *elastic MSM* is based on the vectors of EC points $\mathbf{P}_{ij} := 2^{(j-1)ks}\mathbf{P}_i$, rather than the original vectors of EC points $\mathbf{P}_i$, for $i \in [1,n]$ and $j \in [1,w]$. Fortunately, given points $\{\mathbf{P}_i\}_{i\in[1,n]}$, all the $\{\mathbf{P}_{ij}\}_{i\in[1,n],j\in[1,w]}$ could be pre-computed. Thus, our *elastic MSM* could be regarded as a new modular and adaptive parameter configuration technique, which could be used in all the Pippenger-like algorithms to replace the *windowed MSM* technique. In this way, it could provide a flexible adjustment between the scale of MSM and bit length of scalar, but at the cost of preprocessing the original vectors of EC points.

Preprocessing complexity. Throughout this paper, we will measure memory space overhead in theory by the number of extra points that need to be stored. Given the points $\{\mathbf{P}_i\}_{i\in[1,n]}$, for each $i \in [1,n]$, we need to additionally preprocess and store the points $\left\{2^{jks}\mathbf{P}_i\right\}_{j\in[1,w-1]}$, to obtain all the $\{\mathbf{P}_{ij}\}_{i\in[1,n],j\in[1,w]}$. Thus, such processing requires $(w-1)ks$ PDBLs and $w-1$ extra memory space overhead for each $i \in [1,n]$, and $n(w-1)ks$ PDBLs and $n(w-1)$ extra memory space overhead in total for all $i \in [1,n]$. Therefore, from another perspective, our solution actually provides a trade-off between time and space saving. When our storage space is limited, we can flexibly adjust the storage space requirement from 0 point to all $n\left(\frac{\lambda}{s}-1\right)$ points by adjusting parameter w and corresponding k.

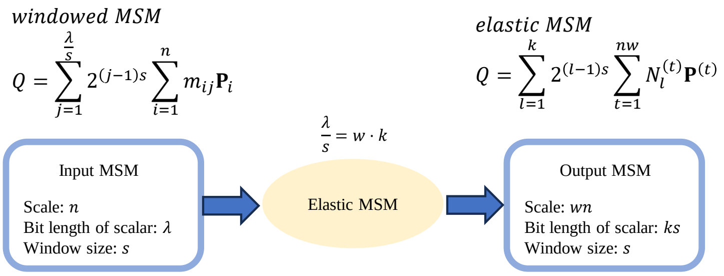

windowed MSM
$Q=\sum_{j=1}^{\frac{\lambda}{s}}2^{(j-1)s}\sum_{i=1}^{n}m_{ij}\mathbf{P}_i$
elastic MSM
$Q=\sum_{l=1}^{k}2^{(l-1)s}\sum_{t=1}^{nw}N_l^{(t)}\mathbf{P}^{(t)}$
$\frac{\lambda}{s}=w\cdot k$
Input MSM
Scale: n
Bit length of scalar: λ
Window size: s
Elastic MSM
Output MSM
Scale: wn
Bit length of scalar: ks
Window size: s

Figure 4: An interpretation of instance conversion.

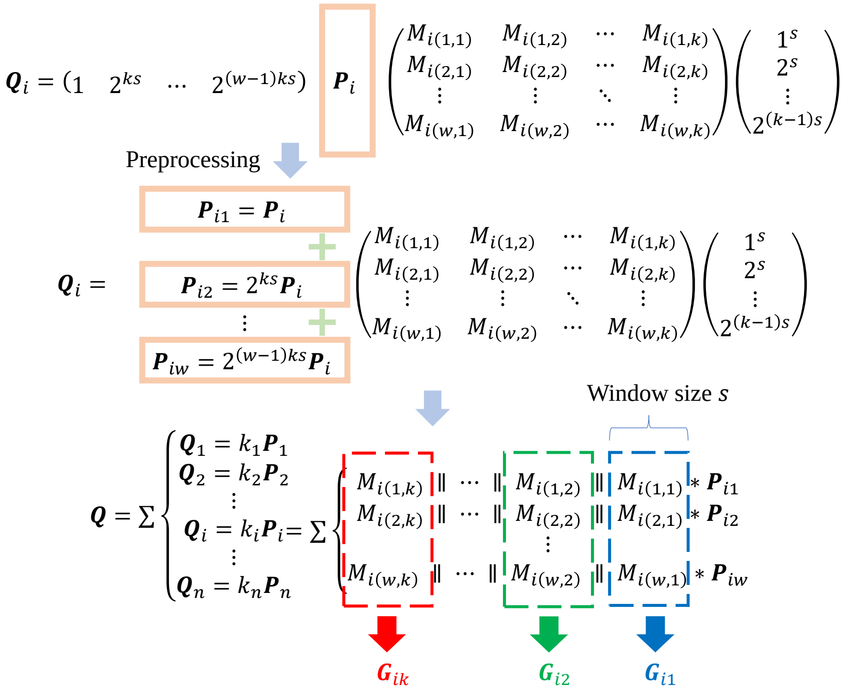


Figure 5: Demonstration of *elastic MSM*

3.2 From The General to The Specific

In the previous subsection, we have proposed our generic *elastic MSM* technique. To deepen our understanding of *elastic MSM*, we will explain its relationship with two specific schemes in this subsection. Recall that with the scale of MSM n, the bit length of scalar λ and window size s, the Pippenger algorithm requires around $\frac{\lambda}{s}\left(n+2^{s+1}\right)$ PADDs plus $\lambda-s$ PDBLs. As we have discussed in **Subsection 3.1**, with $\frac{\lambda}{s}=w\cdot k$, our *elastic MSM* could be used to transform the scale of MSM from n to wn, the bit length of scalar from λ to ks but window size is still s, at the cost of preprocessing requiring $n(w-1)ks$ PDBLs and $n(w-1)$ extra memory space overhead. Then we could conclude that:

Relationship with the *windowed MSM* in Pippenger algorithm. It is obviously that if we set the parameter $w = 1$ in our *elastic MSM*, then there is no preprocessing procedure at all. In this case, by the **Formula (4)**, it is clear that our *elastic MSM* degenerates into the *windowed MSM* in Pippenger algorithm.

Relationship with the *Computation Consolidation* in GZKP. We find it intriguing that the *Computation Consolidation* technique, employed in GZKP [MXS+23], is encompassed as a specific instance within our *elastic MSM* framework. Turning to the other extreme, if we set the the parameter $k = 1$ in our *elastic MSM*, then w reaches its maximum value $\frac{\lambda}{s}$. And the cost of preprocessing also reaches its maximum value $n\left(\frac{\lambda}{s} - 1\right)s$ PDBLs and $n\left(\frac{\lambda}{s} - 1\right)$ extra memory space overhead. However, when *elastic MSM* with $k = 1$ is implemented in the original Pippenger algorithm, there are only $\frac{ks}{s}\left(wn + 2^{s+1}\right) = \frac{\lambda n}{s} + 2^{s+1}$ PADDs and $\lambda - s = ks - s = 0$ PDBLs, which means that we have transferred many computation of PADDs to the preprocessing computation, and completely transferred the cost of PDBLs to the preprocessing computation. This transfer is exactly the same as the *Computation Consolidation* technique used in GZKP. Therefore, following our *elastic MSM*, we may be able to view the idea in *Computation Consolidation* from a higher dimension, and to some extent provide theoretical support for its optimality in cost of both Pippenger PADDs and PDBLs.

In addition to the extreme parameter selection, it is the parameter adjustment in other situations that is the core of our *elastic MSM*, and we will discuss it in detail in **Section 4**. In fact, the GZKP failed to view the preprocessing procedure from a higher perspective like we do, so the preprocessing MSM scheme in GZKP failed to adjust parameters itself, which leads to huge memory space overhead required to store the precomputed points. Though it has been mentioned in GZKP that adjustments of memory space overhead could be achieved through additional combination of a checkpoint strategy, this strategy will incur a significant amount of additional MSM computation.

4 The Elastic Pippenger Algorithm

For the reason that MSM is the most time-consuming operation in zkSNARK, it is very necessary for us to develop the efficient parallel MSM algorithm. And due 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. Thus, in this section, as an example of analysis, we will plug our *elastic MSM* technique into two commonly used parallel Pippenger algorithms to get two new preprocessing parallel Pippenger algorithms—*elastic Pippengers.* Then we will conduct a detailed theoretical comparison between *elastic Pippengers* and the preprocessing parallel Pippenger algorithm obtained by combing the same parallel Pippenger algorithms as *elastic Pippengers* with the preprocessing scheme in GZKP. Note that our *elastic MSM* technique is clearly modular and could be plugged in all Pippenger-like algorithms which include *windowed MSM* to flexibly optimize the original algorithm. Additionally, our *elastic MSM* technique also seems to be compatible with other pre-computation techniques such as the recent one proposed in [LFG23].

4.1 Some Common Parallel Pippenger Algorithms

Based on the original Pippenger algorithm, it is easy to come up with three naive parallel Pippenger algorithms. While the recent work [LWY+23] has summarized them very clearly, for the completeness of our discussion, we will also provide some explanations on these algorithms.

Approach 1. The first naive approach could be regard as a natural derivative of the original Pippenger algorithm. As deduced in the **Formula (1)**, we establish that

$\mathbf{Q} = \sum_{j=1}^{\frac{\lambda}{s}} 2^{(j-1)s}\mathbf{G}_j$, signifying the existence of $\frac{\lambda}{s}$ natural subtasks. So we could just arrange $\frac{\lambda}{s}$ threads to perform these subtasks simultaneously, which however at most provides a speedup of $\frac{\lambda}{s}$. This means that we cannot fully utilize the potential of GPUs for that the $\frac{\lambda}{s}$ is usually much smaller than the number of cores provided by GPUs (λ typically ranges from 254 to 768 and s can be chosen at will).

The complexity of the first naive approach. Just as the original Pippenger algorithm, the first naive approach also converts the task using *windowed MSM* technique first, then however it only need to compute subtask results $\mathbf{G}_j$, for each $j \in \left[1, \frac{\lambda}{s}\right]$ in parallel and at once, these steps require, as analyzed earlier, at most $n + 2^{s+1}$ PADDs. Finally, computing the MSM result with subtask results requires, as analyzed earlier, at most $\lambda - s$ PDBLs and $\frac{\lambda}{s}$ PADDs. Thus, the total first naive approach require at most $n + 2^{s+1} + \frac{\lambda}{s}$ PADDs and $\lambda - s$ PDBLs.

Approach 2. The second naive approach is just a general method for dividing a large-scale task into small-scale modular subtasks. We denote t as the number of total threads (w.l.o.g. we assume that n is divisible by t), then we have $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i = \sum_{j=1}^{t} \mathbf{Q}_j$, where $\mathbf{Q}_j = \sum_{i=1}^{\frac{n}{t}} k_{(j-1)\frac{n}{t}+i}\mathbf{P}_{(j-1)\frac{n}{t}+i}$ where $j \in [1, t]$. So we could perform serial Pippenger algorithms in parallel for all the small-scale MSM corresponding to $\{\mathbf{Q}_j\}_{j\in[1,t]}$. Finally, the result $\mathbf{Q}$ could be computed just with parallel sum algorithm.

The complexity of the second naive approach. We could perform t Pippenger algorithms in parallel and at once for all the small-scale MSM corresponding to $\{\mathbf{Q}_j\}_{j\in[1,t]}$ with λ bits scalar, s bits window size and $\frac{n}{t}$ MSM scale, this computation clearly requires $\frac{\lambda}{s}\left(\frac{n}{t} + 2^{s+1}\right)$ PADDs and $\lambda - s$ PDBLs. Plus $\log t$ PADDs required by parallel sum algorithm, the total second naive approach requires at most $\frac{\lambda}{s}\left(\frac{n}{t} + 2^{s+1}\right) + \log t$ PADDs and $\lambda - s$ PDBLs.

Approach 3. The third naive approach is a combination of the above two approaches. With total t threads, we could use the second approach and then the first approach to decompose the original computation and then perform Pippenger algorithm in parallel. Specifically, we denote the smallest subtask result as $\mathbf{G}_{j,l}$, then we have that

$$\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P}_i = \sum_{j=1}^{t/\frac{\lambda}{s}} \mathbf{Q}_j = \sum_{j=1}^{t/\frac{\lambda}{s}} \sum_{l=1}^{\frac{\lambda}{s}} 2^{(l-1)s}\mathbf{G}_{j,l} \tag{7}$$

The complexity of the third naive approach. According to the **Formula (7)**, we could compute our result $\mathbf{Q}$ by first using the first approach to compute each the small-scale MSM corresponding to $\mathbf{Q}_{j\in[1,t/\frac{\lambda}{s}]}$ (with λ bits scalar, s bits window size and $\frac{n}{t/\frac{\lambda}{s}}$ MSM scale) in parallel, which requires $\frac{n}{t/\frac{\lambda}{s}} + 2^{s+1} + \frac{\lambda}{s} = \frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + \frac{\lambda}{s}$ PADDs and $\lambda - s$ PDBLs. Then we only need to perform parallel sum algorithm to sum all the $\{\mathbf{Q}_j\}_{j\in[1,t/\frac{\lambda}{s}]}$, which requires $\log\left(t/\frac{\lambda}{s}\right)$ PADDs. Thus, the total third naive approach requires at most $\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + \frac{\lambda}{s} + \log\left(t/\frac{\lambda}{s}\right)$ PADDs and $\lambda - s$ PDBLs.

It is clearly that the third naive approach performs better than other two approaches in the case of high parallelism in theory. However, it is worth noting that compared to the third approach, the second approach could be optimized more by preprocessing techniques. Therefore, the second approach is more worthy of preprocessing preprocessing and often used with preprocessing techniques. In fact, if combined with preprocessing techniques, the second approach may be as fast as the third approach in implementation. So in our next discussion, we will consider both these approaches simultaneously. Form now on, we call the second approach and the third approach just *fast parallel Pippenger 2* and *fast parallel Pippenger 3* respectively. Although there have been faster algorithms recently such as the scheme in [LWY+23], to provide a more concise explanation of the advantages that our *elastic MSM* technique will bring, we will only plug our *elastic MSM* into *fast* *parallel Pippenger 2 (3)* to obtain our *elastic Pippenger 2 (3)*, and analyze them as an example. In fact, Since our *elastic MSM* is a modular technique, it could also be plugged into any similar algorithm which has *windowed MSM* technique for speed improvement, including the scheme in [LWY+23].

4.2 About the Checkpoint Strategy Used in GZKP

From our discussed in **Subsection 3.2**, as a special case of *elastic MSM* with $k = 1$, the *Computation Consolidation* technique proposed in GZKP [MXS+23] seems to be optimal in terms of MSM computation. However, as has also been mentioned in GZKP, the memory space overhead required to store the pre-computed points is over 5 GB at the MSM scale of 2^{21}, and further grows as the MSM scale increases. Thus to endow the *Computation Consolidation* technique with the ability to adapt to different space limitations, GZKP also proposed a *Checkpoints* technique to better balance between the time and space saving. For the convenience of our subsequent comparison, we will first introduce the *Checkpoints* technique in this subsection.

As we have discussed in **Subsection 3.2**, while GZKP could reduce the MSM computation, the cost of preprocessing of GZKP reaches its maximum value $n(\frac{\lambda}{s} - 1)ks$ PDBLs and $n(\frac{\lambda}{s} - 1)$ extra memory space overhead, fortunately, *Checkpoints* technique proposed in GZKP shows that we could only preprocess and store the points with fixed weights, called checkpoints, i.e. $(2^{M*s}, 2^{(2M)*s}, \ldots, 2^{\left\lfloor \frac{\lambda/s}{M} \right\rfloor * M * s}) * \mathbf{P}_i$ for $i \in [1, n]$, rather than all the points $(2^s, 2^{2*s}, \ldots, 2^{(\frac{\lambda}{s}-1)*s}) * \mathbf{P}_i$ for $i \in [1, n]$, where $1 \leq M < \frac{\lambda}{s}$ is an integer interval. So, when we sum all points in the bucket, for each point, we could first find the closest checkpoint to the needed point, and then perform at most $(M - 1)\,s$ PADDs to get the desired point. If t is the number of total threads, It is clear that the total additional preprocessing cost brought by *Checkpoints* technique is at most $n \left\lfloor \frac{\lambda/s}{M} \right\rfloor$ extra memory space overhead, $\frac{n}{t} \left\lfloor \frac{\lambda/s}{M} \right\rfloor Ms$ PDBLs. And the total additional MSM computation cost brought by *Checkpoints* technique is at most $\frac{n}{t}(\frac{\lambda}{s} - 1)(M - 1)s$ PADDs. For the convenience of theoretical comparison, we summarize in **Table 1** the additional overhead brought by *Checkpoints* technique and our *elastic MSM* technique both in parallel setting.

Table 1: Additional preprocessing overhead comparison

	Space Overhead	Extra PDBLs (Prep)	Extra PADDs (MSM)
Checkpoints technique	$n \left\lfloor \frac{\lambda/s}{M} \right\rfloor$	$\frac{n}{t} \left\lfloor \frac{\lambda/s}{M} \right\rfloor Ms$	$\frac{n}{t}(\frac{\lambda}{s} - 1)(M - 1)s$
elastic MSM technique	$n(w - 1)$	$\frac{n}{t}(w - 1)ks$	0

This table shows us the extra overhead brought by *Checkpoints* technique and our *elastic MSM* technique both in parallel setting. We use (Prep) and (MSM) to represent the extra preprocessing overhead and the extra MSM computation overhead respectively. n is the MSM scale, t is the number of total threads, λ is the bit length of scalar, s is the window size, the flexible parameter M is an integer interval and the flexible parameter w, k satisfy $\frac{\lambda}{s} = w \cdot k$. The memory space overhead in theory is measured by the number of EC points that need to be additionally stored.

So far, it seems that while our *elastic MSM* technique could be used directly to balance between the time and space saving by adjusting the flexible parameter w, k, the *Computation Consolidation* technique and the *Checkpoints* technique both proposed in GZKP could also be used together to achieve the same thing by choosing appropriate M. However, we need to emphasize that in order to achieve the same goal (in other words, computing MSM under the same storage space limitations), the additional computational cost of our solution is much smaller than that of GZKP's solution as we will discuss next.

4.3 Theoretical Analysis and Comparison

From the discussion of the previous subsections, we could know that the preprocessing technique *Computation Consolidation* itself does not contain any flexible parameter, and when we plug it into the Pippenger-like parallel algorithms (we use the *fast parallel Pippenger 2 (3)* as an example in this paper), it should be combined with *Checkpoints* technique to support balance between the time and space saving. And we call *fast parallel Pippenger 2* and *fast parallel Pippenger 3* combined with both the *Computation Consolidation* technique and *Checkpoints* technique *GZKP Pippenger 2* and *GZKP Pippenger 3* respectively. Recall that our preprocessing technique *elastic MSM* itself could be regarded as a modular and adaptive parameter configuration technique, so it is a plug-and-play technique, that is when we plug the *elastic MSM* into the Pippenger-like parallel algorithms (the *fast parallel Pippenger 2 (3)* as an example in this paper), we could directly obtain algorithms *elastic Pippenger 2* and *elastic Pippenger 3*, which support balance between the time and space saving. In this subsection, we will show the theoretical analysis about various complexities of *GZKP Pippenger 2 (3)* and *elastic Pippenger 2 (3)*, then we will evaluate and compare these two algorithms in theory under some practical parameters.

Algorithm with approach 2. Recall that we have introduced in **Subsection 4.1**, to compute MSM, *fast parallel Pippenger 2* requires at most $\frac{\lambda}{s}\left(\frac{n}{t}+2^{s+1}\right)+\log t$ PADDs and $\lambda - s$ PDBLs. As has been discussed in **Subsection 3.1**, to compute the MSM with n scale, λ bit length scalar and s window size, our *elastic MSM* could be used to first transform this MSM instance into another with wn scale, ks bit length scalar and s window size, where $\frac{\lambda}{s} = w \cdot k$. So the *elastic Pippenger 2* algorithm requires $\frac{\lambda}{s} \cdot \frac{n}{t} + k \cdot 2^{s+1} + \log t$ PADDs and $ks - s$ PDBLs to compute the MSM. From the **Subsection 3.2**, we could know that the *Computation Consolidation* technique is included as a special case in our *elastic MSM* ($k = 1$). Thus, the *GZKP Pippenger 2* algorithm (regardless of *Checkpoints* technique) requires $\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + \log t$ PADDs and 0 PDBLs to compute the MSM. Now, we summarize in **Table 2** the overall theoretical complexity of the *GZKP Pippenger 2* algorithm and the *elastic Pippenger 2* algorithm.

Table 2: Comparison between *GZKP Pippenger 2* and *elastic Pippenger 2*

	Space Overhead	Precomputing PDBLs	MSM PDBLs	MSM PADDs
GZKP Pippenger 2	$n\left\lfloor\frac{\lambda/s}{M}\right\rfloor$	$\frac{n}{t}\left\lfloor\frac{\lambda/s}{M}\right\rfloor Ms$	0	$\frac{\lambda}{s}\cdot\frac{n}{t}+2^{s+1}+\log(t)+\frac{n}{t}(\frac{\lambda}{s}-1)(M-1)s$
elastic Pippenger 2	$n(w-1)$	$\frac{n}{t}(w-1)ks$	$ks-s$	$\frac{\lambda}{s}\cdot\frac{n}{t}+k\cdot 2^{s+1}+\log(t)$

This table shows us overall theoretical complexity of the *GZKP Pippenger 2* algorithm and the *elastic Pippenger 2* algorithm. n is the MSM scale, t is the number of total threads, λ is the bit length of scalar, s is the window size, the flexible parameter M is an integer interval and the flexible parameter w, k satisfy $\frac{\lambda}{s} = w \cdot k$. The memory space overhead in theory is measured by the number of EC points that need to be additionally stored.

At first glance, compared to the *GZKP Pippenger 2*, it is easy for us to see from the **Table 2** that our *elastic Pippenger 2* sacrificed a very small amount of computational efficiency for MSM PDBLs and also a amount of computational efficiency for MSM PADDs (regardless the $\frac{n}{t}(\frac{\lambda}{s}-1)(M-1)s$ MSM PADDs brought by GZKP preprocessing). However, these sacrifices are relatively small compared to the additional MSM PADDs overhead brought by GZKP preprocessing.

Algorithm with approach 3. Recall that we have introduced in **Subsection 4.1**, to compute MSM, *fast parallel Pippenger 3* requires at most $\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + \frac{\lambda}{s} + \log\left(t/\frac{\lambda}{s}\right)$ PADDs and $\lambda - s$ PDBLs. As has been discussed in **Subsection 3.1**, to compute the MSM with n scale, λ bit length scalar and s window size, our *elastic MSM* could be used to first transform this MSM instance into another with wn scale, ks bit length scalar and s window size, where $\frac{\lambda}{s} = w \cdot k$. So the *elastic Pippenger 3* algorithm requires $\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + k + \log\left(\frac{t}{k}\right)$

PADDs and $ks - s$ PDBLs to compute the MSM. From the **Subsection 3.2**, we could know that the *Computation Consolidation* technique is included as a special case in our *elastic MSM* ($k = 1$). Thus, the *GZKP Pippenger 3* algorithm (regardless of *Checkpoints* technique) requires $\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + 1 + \log(t)$ PADDs and 0 PDBLs to compute the MSM. Now, we summarize in **Table 3** the overall theoretical complexity of the *GZKP Pippenger 3* algorithm and the *elastic Pippenger 3* algorithm.

Table 3: Comparison between *GZKP Pippenger 3* and *elastic Pippenger 3*

	Space Overhead	Precomputing PDBLs	MSM PDBLs	MSM PADDs
GZKP Pippenger 3	$n\left\lfloor\frac{\lambda/s}{M}\right\rfloor$	$\frac{n}{t}\left\lfloor\frac{\lambda/s}{M}\right\rfloor Ms$	0	$\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + 1 + \log(t) + \frac{n}{t}(\frac{\lambda}{s} - 1)(M-1)s$
elastic Pippenger 3	$n(w-1)$	$\frac{n}{t}(w-1)ks$	$ks - s$	$\frac{\lambda}{s} \cdot \frac{n}{t} + 2^{s+1} + k + \log\left(\frac{t}{k}\right)$

This table shows us overall theoretical complexity of the *GZKP Pippenger 3* algorithm and the *elastic Pippenger 3* algorithm. n is the MSM scale, t is the number of total threads, λ is the bit length of scalar, s is the window size, the flexible parameter M is an integer interval and the flexible parameter w, k satisfy $\frac{\lambda}{s} = w \cdot k$. The memory space overhead in theory is measured by the number of EC points that need to be additionally stored.

At first glance, compared to the *GZKP Pippenger 3*, it is easily for us to see from the **Table 3** that we sacrificed a very small amount of computational efficiency for MSM PDBLs in exchange for a huge improvement in MSM PADDs efficiency. Moreover, it is clear that the overhead of MSM computation in *GZKP Pippenger 3* is almost the same as that in *GZKP Pippenger 2*. This shows that the second approach could be optimized more by preprocessing techniques for that the third approach performs better than the second approach in the case of just parallelism.

We assume that the size of the storage space is only enough to accommodate Q extra points, according to the **Table 3** we have that $n(w-1) = Q$, then $w = \frac{Q}{n} + 1, k = \frac{\lambda}{s(Q/n+1)}$. Since w and k together represent the dimension of a matrix, both of them are at least equal to or greater than 1. Then we have $\frac{\lambda}{s(Q/n+1)} \geq 1$, that is $Q \leq \frac{n\lambda}{s} - n$. In fact, if the inequality $Q \geq \frac{n\lambda}{s} - n$ holds, then there is enough space to store all the $n\left(\frac{\lambda}{s} - 1\right)$ points, in which case our *elastic MSM* technique will degenerate into the *Computation Consolidation* technique. Next, we will use some specific parameters to conduct a very rough theoretical evaluation of these two schemes. If we set the parameter $n = 2^{22}, t = 2^{12}, \lambda = 3 \cdot 2^8, s = 2^3, Q = 7 \cdot 2^{22}$, then we have the following evaluation:

We first consider the *GZKP Pippenger 2* and *GZKP Pippenger 3*, from $n\left\lfloor\frac{\lambda/s}{M}\right\rfloor = Q$ we could know that the minimum M that satisfies this equation is 13. Then the check points are $\{2^{13s}, 2^{26s}, 2^{39s}, 2^{52s}, 2^{65s}, 2^{78s}, 2^{91s}\} * \mathbf{P}_i$ for $i \in [1, n]$, the parallel pre-computation of which requires $\frac{n}{t}\left\lfloor\frac{\lambda/s}{M}\right\rfloor Ms = 91 \cdot 2^{13}$ PDBLs. During the MSM computation, the points $\{2^s, 2^{2s}, \ldots, 2^{95s}\} * \mathbf{P}_i$ for $i \in [1, n]$ need to be additionally computed from the check points by $\frac{n}{t} \cdot 304s = 19 \cdot 2^{17}$ PADDs. So, it is clear that the MSM computation in *GZKP Pippenger 2* needs a total of $19 \cdot 2^{17} + 3 \cdot 2^{15} + 2^9 + 12 \approx 79 \cdot 2^{15}$ PADDs. And the MSM computation in *GZKP Pippenger 3* needs a total of $19 \cdot 2^{17} + 3 \cdot 2^{15} + 2^9 + 1 + 12 \approx 79 \cdot 2^{15}$ PADDs.

Then we consider the *elastic Pippenger 2* and *elastic Pippenger 3*, from $Q = 7 \cdot 2^{22}$ we know that $w = \frac{Q}{n} + 1 = 8$, then $k = \frac{\lambda/s}{w} = 12$. Thus, the preprocessing of *elastic Pippenger 2 (3)* requires $\frac{n}{t}(w-1)ks = 21 \cdot 2^{15}$ PDBLs. While the *elastic Pippenger 2 (3)* requires $ks - s = 11 \cdot 2^3$ MSM PDBLs, the MSM PADDs required by *elastic Pippenger 2* and *elastic Pippenger 3* are just $3 \cdot 2^{15} + 3 \cdot 2^{11} + 12 \approx 3 \cdot 2^{15}$ and $3 \cdot 2^{15} + 2^9 + 12 + 10 - \log 3 \approx 3 \cdot 2^{15}$ respectively.

Using the same evaluation method as the above example, we compare the theoretical results of the *GZKP Pippenger 2 (3)* and the *elastic Pippenger 2 (3)* under different storage space limitations. Due to our rough theoretical estimation, theoretical results about approach 2 and approach 3 are the same. Then, we present them in **Table 4**. It is clear that the additional MSM overhead caused by the preprocessing in *GZKP Pippenger 2 (3)* is unacceptable. Compared to the *GZKP Pippenger 2 (3)*, it is obvious that our *elastic Pippenger 2 (3)* requires less PDBLs during the preprocessing. Moreover, our *elastic Pippenger 2 (3)* requires significantly less MSM PADDs while only increasing the very few extra MSM PDBLs cost. Note that as a norm in Pippenger-like algorithms, if we assume for simplicity the computational cost of PDBLs and that of PADDs in EC are the same, it is clear that extra MSM PDBLs cost in our *elastic Pippenger* could be just ignored.

Table 4: Comparison of theoretical valuations for *GZKP Pippenger* and *elastic Pippenger*.

	GZKP Pippenger 2 (3)			*elastic Pippenger 2 (3)*		
Extra Space Overhead	Precomputing PDBLs	MSM PDBLs	MSM PADDs	Precomputing PDBLs	MSM PDBLs	MSM PADDs
$7 \cdot 2^{22}$	$22.75 \cdot 2^{15}$	0	$79 \cdot 2^{15}$	$21 \cdot 2^{15}$	$11 \cdot 2^{3}$	$3 \cdot 2^{15}(26.3\times)$
$5 \cdot 2^{22}$	$21.25 \cdot 2^{15}$	0	$106.75 \cdot 2^{15}$	$20 \cdot 2^{15}$	$15 \cdot 2^{3}$	$3 \cdot 2^{15}(35.6\times)$
$3 \cdot 2^{22}$	$18.75 \cdot 2^{15}$	0	$172.5 \cdot 2^{15}$	$18 \cdot 2^{15}$	$23 \cdot 2^{3}$	$3 \cdot 2^{15}(57.5\times)$
$2 \cdot 2^{22}$	$16.5 \cdot 2^{15}$	0	$179.75 \cdot 2^{15}$	$16 \cdot 2^{15}$	$31 \cdot 2^{3}$	$3 \cdot 2^{15}(59.9\times)$
2^{22}	$12.25 \cdot 2^{15}$	0	$423.25 \cdot 2^{15}$	$12 \cdot 2^{15}$	$47 \cdot 2^{3}$	$3 \cdot 2^{15}(141.1\times)$

This table shows us the theoretical performance comparison under different preprocessing storage space limitations when we set the parameter $n = 2^{22}, t = 2^{12}, \lambda = 3 \cdot 2^{8}, s = 2^{3}$.

5 Evaluation

In this section, we first give our experimental setting in **Subsection 5.1**. Then, we show some experimental results of both *GZKP Pippenger 2 (3)* and *elastic Pippenger 2 (3)* in **Subsection 5.2** to highlight the improvement that our preprocessing parallel MSM algorithm provides.

5.1 Methodology

Our experiments are conducted on a server with daul 2.20 GHz Inter Xeon E5-2630 processors and 128 GB DRAM, running Ubuntu 20.04.6 with CUDA Toolkit 11.6. It is equipped with one NVIDIA GeForce GTX 2080Ti GPU card (with 11 GB memory). Its CPU-GPU data transfer is completed through PCI-E. Recognize that distinctions in hardware resources may exert a substantial influence on the outcomes of comparisons. Consequently, to facilitate comparisons with a degree of fairness, we meticulously orchestrate the evaluation of GPU implementations within a uniform testbed environment. Notably, just moving EC points from host memory to device memory takes much time on data transfer. Thus we overlap the CPU-GPU data transfer and device computing based on the multi-streaming technique during our experiments.

5.2 Performance

Here, we examine and compare the execution time of the preprocessing parallel MSM algorithms *GZKP Pippenger 2 (3)* and *elastic Pippenger 2 (3)*. We evaluate all these algorithms on the dense synthetic data created by libsnark [lib14] with the 753-bit MNT4753 curve.

Table 5 and **Table 6** provide the evaluation results on the precomputing time and MSM time of *GZKP Pippenger 2 (3)* and *elastic Pippenger 2 (3)* across various preprocessing storage space limitations. Given a range of practical parameters, it can be concluded from these tables that our *elastic Pippenger 2 (3)* has a slight advantage in precomputing time compared to the *GZKP Pippenger 2 (3)* across almost all preprocessing storage space limitations. Notably, the precomputing time of *GZKP Pippenger 2* or *elastic Pippenger 2* is the same as the precomputing time of *GZKP Pippenger 3* or *elastic Pippenger 3*. This is because the only difference lies in the use of different MSM algorithms, but the preprocessing methods are consistent.

Additionally, the difference in MSM time between *GZKP Pippenger 2* and *GZKP Pippenger 3* is very small. This is because theoretically, their computational complexity are almost the same. However, the MSM time of the *elastic Pippenger 2* is obviously longer than that of the *elastic Pippenger 3*, due to significant differences in computational complexity. When the storage space is limited to storing $7 \cdot 2^{22} - 2^{22}$ extra points, our *elastic Pippenger 2* achieves about $4 - 28\times$ speedup versus the *GZKP Pippenger 2* and our *elastic Pippenger 3* achieves about $11 - 45\times$ speedup versus the *GZKP Pippenger 3*. Therefore, under the same storage space limitations, our solution has significant efficiency advantages. From another perspective, we can infer that under the same time constraints, our solution requires less storage space. Furthermore, the stricter the restrictions on storage space, the more advantageous our construction is.

Table 5: Performance results (in seconds) about approach 2 and across various storage space limitations.

	GZKP Pippenger 2		*elastic Pippenger 2*	
Extra Space Overhead	Precomputing time	MSM time	Precomputing time	MSM time
$7 \cdot 2^{22}$	19.215	1.167	17.691	0.290(4.0×)
$5 \cdot 2^{22}$	13.604	4.783	12.942	0.313(15.3×)
$3 \cdot 2^{22}$	8.322	8.289	8.053	0.325(25.5×)
$2 \cdot 2^{22}$	5.744	10.148	5.609	0.414(24.5×)
2^{22}	3.214	12.134	3.149	0.431(28.2×)

This table shows us the experimental performance comparison under different preprocessing storage space limitations when we set the parameter $n = 2^{22}, t = 2^{12}, \lambda \approx 3 \cdot 2^8, s = 2^3$.

Table 6: Performance results (in seconds) about approach 3 and across various storage space limitations.

	GZKP Pippenger 3		*elastic Pippenger 3*	
Extra Space Overhead	Precomputing time	MSM time	Precomputing time	MSM time
$7 \cdot 2^{22}$	19.215	1.087	17.691	0.092(11.8×)
$5 \cdot 2^{22}$	13.604	4.837	12.942	0.161(30.0×)
$3 \cdot 2^{22}$	8.322	8.285	8.053	0.203(40.8×)
$2 \cdot 2^{22}$	5.744	10.080	5.609	0.230(43.8×)
2^{22}	3.214	12.104	3.149	0.268(45.2×)

This table shows us the experimental performance comparison under different preprocessing storage space limitations when we set the parameter $n = 2^{22}, t = 2^{12}, \lambda \approx 3 \cdot 2^8, s = 2^3$.

Table 7 and **Table 8** give the precomputing time and MSM time of *GZKP Pippenger 2 (3)* and *elastic Pippenger 2 (3)* across various MSM scales. The parameters t, λ and s selected in this experiment are consistent with the parameters used in the previous experiment. We chose a moderate storage space limit in the previous experiment, which means that we fix $Q = 3 \cdot 2^c$, *integer* $c \in [16, 22]$ in this experiment (note that Q should change with n to be meaningful). It can be concluded from these tables that our *elastic Pippenger 2 (3)* has a slight advantage in precomputing time compared to the *GZKP Pippenger 2 (3)* across almost all MSM scales. As we have discussed above, the precomputing time of *GZKP Pippenger 2* or *elastic Pippenger 2* is the same as the precomputing time of *GZKP Pippenger 3* or *elastic Pippenger 3*, the MSM time of the *GZKP Pippenger 2* is almost the same as that of the *GZKP Pippenger 3*, and the MSM time of the *elastic Pippenger 2* is obviously longer than that of the *elastic Pippenger 3*.

Additionally, when the MSM scale is limited to $2^{16} - 2^{22}$, our *elastic Pippenger 2* achieves about $10 - 25\times$ speedup versus the *GZKP Pippenger 2* and our *elastic Pippenger 3* achieves about $14 - 40\times$ speedup versus the *GZKP Pippenger 3*. Therefore, the larger the scale of MSM, the more advantageous our constructions are.

Table 7: Performance Results (in seconds) about approach 2 and across various MSM scales.

	GZKP Pippenger 2		*elastic Pippenger 2*	
MSM Scale	Precomputing time	MSM time	Precomputing time	MSM time
2^{16}	0.178	0.220	0.173	0.021(10.5×)
2^{17}	0.328	0.371	0.314	0.026(14.3×)
2^{18}	0.563	0.698	0.528	0.045(15.5×)
2^{19}	1.091	1.392	1.050	0.074(18.8×)
2^{20}	2.081	2.029	2.024	0.117(17.3×)
2^{21}	4.181	4.162	4.004	0.191(21.8×)
2^{22}	8.322	8.289	8.053	0.325(25.5×)

This table shows us the experimental performance comparison under different MSM scales when we set the parameter $t = 2^{12}, \lambda \approx 3 \cdot 2^8, s = 2^3$.

Table 8: Performance Results (in seconds) about approach 3 and across various MSM scales.

	GZKP Pippenger 3		*elastic Pippenger 3*	
MSM Scale	Precomputing time	MSM time	Precomputing time	MSM time
2^{16}	0.178	0.216	0.173	0.015(14.4×)
2^{17}	0.328	0.361	0.314	0.021(17.2×)
2^{18}	0.563	0.691	0.528	0.034(20.3×)
2^{19}	1.091	1.374	1.050	0.055(25.0×)
2^{20}	2.081	2.018	2.024	0.105(19.2×)
2^{21}	4.181	4.151	4.004	0.170(24.4×)
2^{22}	8.322	8.285	8.053	0.203(40.8×)

This table shows us the experimental performance comparison under different MSM scales when we set the parameter $t = 2^{12}, \lambda \approx 3 \cdot 2^8, s = 2^3$.

References

[ABC+22] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. FPGA acceleration of multi-scalar multiplication:

Cyclonemsm. *IACR Cryptol. ePrint Arch.*, page 1396, 2022.

[BBB$^+$18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

[BCI$^+$13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 315–333. Springer, 2013.

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

[Bel19] Bellperson: Gpu parallel acceleration for zksnark, 2019. https://github.com/filecoin-project/bellperson, Accessed: 2023-10-10.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 103–112, New York, 1988. ACM.

[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. *IACR Cryptol. ePrint Arch.*, page 352, 2020.

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

[CHM$^+$20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, 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, 2003.

[DLFKP16] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 235–254, 2016.

[dR95] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 389–399, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

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

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 321–340, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

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

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.

[GY19] Hisham S. Galal and Amr M. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 265–278, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

[Har22] Hardcaml zprize, 2022. https://zprize.hardcaml.com/, Accessed: 2023-10-10.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 723–732, New York, 1992. ACM.

[LFG23] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):358–380, 2023.

[lib14] libsnark: a c++ library for zksnark proofs, 2014. https://github.com/scipr-lab/libsnark, Accessed: 2023/12/1.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In Ronald Cramer, editor, *Theory of Cryptography*, pages 169–189, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[LWY+23] Tao Lu, Chengkun Wei, Ruijing Yu, Yi Chen, L. xilinx Wang, Chaochao Chen, Zeke Wang, and Wenzhi Chen. cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023:194–220, 2023.

[Mat22] Accelerating msm operations on gpu/fpga, 2022. https://github.com/matter-labs/z-prize-msm-gpu, Accessed: 2023-10-10.

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

[Min19] Mina: gpu groth16 prover, 2019. https://github.com/MinaProtocol/gpu-groth16-prover-3x, Accessed: 2022-10-10.

[MXS+23] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. Gzkp: A gpu accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 340–353, New York, NY, USA, 2023. Association for Computing Machinery.

[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, 1976.

[Spp22] Zero-knowledge template library, 2022. https://github.com/supranational/sppark, Accessed: 2023-10-10.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. Dizk: A distributed zero knowledge proof system. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 675–692, USA, 2018. USENIX Association.

[Xav22] Charles. F. Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. Cryptology ePrint Archive, Paper 2022/999, 2022. https://eprint.iacr.org/2022/999.

[Yrr22] Z-prize msm on the gpu submission, 2022. https://github.com/yrrid/submission-msm-gpu, Accessed: 2023-10-10.

[ZC16] Zhichao Zhao and T.-H. Hubert Chan. How to vote privately using bitcoin. In Sihan Qing, Eiji Okamoto, Kwangjo Kim, and Dongmei Liu, editors, *Information and Communications Security*, pages 82–96, Cham, 2016. Springer International Publishing.

[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*, CCS '20, page 2039–2053, New York, NY, USA, 2020. Association for Computing Machinery.

[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, 2017.

[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, 2021.