Computing Multi-Scalar Multiplication on memory-constrained devices*

Léo Noël[1] and Thomas Plantard[2]

[1]Nokia Bell labs, 12 rue Jean Bart, Massy, 91300, France.
[1]Université de Bordeaux, CNRS, INRIA, IMB, UMR 5251, Talence, 33400, France.
[2]Nokia Bell labs, 600-700 Mountain Avenue, Murray Hill, 07974-0636, USA.

Contributing authors: leo.noel@nokia-bell-labs.com;
thomas.plantard@nokia-bell-labs.com;

Abstract

Multi-Scalar Multiplication is a critical operation in most pairing-based zero-knowledge proofs. In a lot of studies, memory limitations have often been reported to be the primary bottleneck preventing the calculation of larger MSMs. In this paper, we are particularly interested in the acceleration of this operation on devices with limited memory.
Pippenger's algorithm (also known as bucket method) is the most efficient and, consequently, the most widely used method to calculate Multi-Scalar Multiplications. We propose an optimization of Pippenger's algorithm which is at least as efficient as the original, and significantly more effective when operating under limited memory. The main idea is to use an adapted number of buckets depending on the available memory instead of $\mathbf{2^w - 1}$. We conducted tests on the curve BLS12-381 with Multi-Scalar Multiplications ranging from $\mathbf{2^8}$ to $\mathbf{2^{14}}$ points. The results obtained demonstrate that we have a very significant gain (up to **40%**) for very limited memories. This gain gradually decreases as more memory becomes available, until we achieve performance comparable to Pippenger's once memory is no longer limited. For example, in a Multi-Scalar Multiplication with $\mathbf{2^{13}}$ points, we observe a gain of **40%** with only **1** KB of memory, **20%** with **15** KB, **15%** with **35** KB, and so on, down to be equivalent to Pippenger's algorithm once memory is no longer a constraint.

Keywords: Multi-scalar Multiplication, Pippenger algorithm, Bucket method, zk-SNARK

---

*This is the author's accepted version of a paper published in Journal of Cryptographic Engineering (11 May 2026, Volume 16, article number 7) available at: https://link.springer.com/article/10.1007/s13389-026-00393-z

Acknowledgments

The authors would like to thank Guilhem Castagnos from the University of Bordeaux, for his careful review of the manuscript and for the valuable comments that helped improve this work.

1 Introduction

The concept of zero-knowledge proofs (ZKPs) was introduced by Goldwasser, Micali, and Rackoff in 1985. It allows one party (the prover) to convince another party (the verifier) that a statement is true without revealing any additional information. One of the most practical and efficient variants of ZKPs is the zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) [7]. Zk-SNARKs offer short proofs and quick verification times, making them well-suited for a wide range of applications, including blockchain technologies [6, 15, 3] and verifiable computation [25, 8].

Today, most zk-SNARKs constructions rely on elliptic curves for both proof generation and verification. While highly efficient, zk-SNARKs still impose significant computational demands, particularly during the proof generation phase. The primary bottleneck in the proving process is the Multi-Scalar Multiplication (MSM) operation [14, 24, 1], which accounts for approximately 70% of the total proving time. Given n points $P_1, \cdots, P_n$ and n scalars $a_1, \cdots, a_n$, MSM is the operation $\sum_{i=1}^{n} a_i P_i$. In zk-SNARKs applications, we are interested in MSM with $n \geq 2^{10}$ [24]. As we just said, MSM often dominates the cost of zero-knowledge proof generation and verification, particularly in zk-SNARK and pairing-based systems. This is why optimizations of MSM are a major focus of recent research [14, 24, 23]. While most state-of-the-art algorithms, such as Pippenger's algorithm [26], achieve excellent asymptotic performance, the amount of available memory (principally dedicated to the storing of intermediate buckets) strongly limits the overall efficiency of these algorithms on memory-constrained devices. Research are conducted to implement cryptographic proofs on memory-limited embedded and IoT systems, and research prototypes show that zero-knowledge proofs can be executed on constrained hardware with careful optimizations of the algorithms [28]. In such environments, where available working memory may be only tens of kilobytes, Pippenger's algorithm either must severely reduce its window (and then its bucket number) or cannot be used at all. If the MSMs are relatively small, algorithms such as Straus's algorithm [31] or the Bos-Coster algorithm [18] can be used because they don't require too much memory [12]. However, for larger MSMs, Pippenger's algorithm is generally preferable [11]. The algorithm we propose in this paper could therefore enable Pippenger's algorithm in ZKPs on devices that previously could not support it, for example, lightweight embedded systems performing proofs.

1.1 Related results

The most natural method for performing scalar multiplication of a point on an elliptic curve is the double-and-add algorithm. However, in the context of Multi-Scalar Multiplication, the bucket method introduced by Pippenger [26] remains by far the most

efficient and widely adopted approach. The core idea of this algorithm is to decompose the scalars into w-bit windows and then assign the corresponding points to buckets according to the value of the w-bits. w is a free parameter left to the user's choice. The larger w is, the more buckets there will be $(2^w - 1)$, and consequently, the more memory will be used. It is also important to note that for each MSM size, there is an optimal window $2 \leq w \leq log_2(n)$ (where n is the number of points in the MSM) that minimizes the number of operations on the curve. Finally, a double-and-add type procedure is applied to combine the contributions of all buckets, yielding the desired result.

In order to improve the efficiency of this algorithm, several directions have been explored. First, different representations of elliptic curves (such as the twisted Edwards form, for example), and the use of alternative coordinate systems (e.g., Jacobian, projective, or extended coordinates) can significantly decrease the number of required field operations, for instance, by avoiding costly modular inversions. However, some of these representations entail considerable memory requirements in the context of large-scale MSMs. For example, while extended coordinates on twisted Edwards curves [9] allow faster arithmetic on the curve, they require the storage of four field elements to represent a single point, compared to only three in projective coordinates. Then, a lot of very clever optimizations have been proposed [14, 24, 22]. One of the first was to use a signed representation of the scalars to reduce the number of buckets by half in each window. By doing that, it allows to reduce significantly the memory used during the algorithm and also the complexity of the bucket aggregation step.

Nevertheless, when performing MSMs over millions of points, memory consumption remains the primary bottleneck. Achieving state of the art performance while significantly reducing memory usage constitutes the central focus of our work. As we show in section 3.2.4, the first naive approach of Pippenger's algorithm implied to generate all the $\lceil \frac{b}{w} \rceil \times (2^w - 1)$ buckets from the beginning. It was obviously not the better idea, so the first idea to reduce memory footprint is to generate the sets of buckets in succession. For each window, only materialize the buckets that actually receive contributions, process them, and reuse the same memory for the next window. By doing that, the temporary memory scales with the number of non-empty buckets instead of the theoretical maximum: $2^w - 1$ buckets. Then the idea of using 2-NAF representation emerged. Using it to represent directly the scalars was useless, but using it after partitioning the scalars into w-bits allows to divide the number of buckets by two and then, save a lot of memory (from $2^w - 1$ to 2^{w-1} buckets for each window). The trick is explained more in details in section 3.2.6. These two variants are totally compatibles with parallelization, just as our proposition, explained in the following section.

1.2 Our contribution

As we just said, memory optimization appears to be a critical factor in the execution of large-scale MSMs. As noted in the Abstract, in many studies [24, 27, 14], memory limitations have often been reported to be the primary constraint preventing the calculation of larger MSMs. This is why we present a variant of Pippenger's algorithm (AdaptiveSetBucket, Algorithm 4), which is more efficient than the original one for

computing MSMs on devices with limited memory. The main idea is that if we cannot choose the optimal window w because storing $2^w - 1$ buckets exceeds the memory capacity, we instead use a smaller number of buckets that do not depend on w. To perform our tests, we use the curve BLS12-381 as it is a widely used elliptic curve in zk-SNARK constructions. Based on the RELIC library [2] and with points in projective coordinates, we achieve gains of up 40% in certain cases, particularly under severely constrained memory conditions. However, it is important to note that our optimization is independent of the chosen curve or the coordinate system of the points. We also tested our algorithm with other coordinate systems (for example, extended coordinates on twisted Edwards curves) and obtained very similar results to those presented in section 4.2. These parameters were selected only to allow comparison with other implementations. For the presentation of our results, we did not account for the storage of points and scalars, since this cost is identical across all implementations. Instead, we only considered the memory consumed within the algorithm itself, primarily for storing the buckets. Moreover, as some studies report that points can be stored in a separate memory space [1], we chose this approach to provide more representative results.

1.3 Outline of the paper

First, we recall some definitions about elliptic curves in Subsection 2.1 and give a formal definition of Multi-Scalar Multiplication in Subsection 2.2. Then we move to Section 3, where we give a quick explanation of Straus's algorithm and a complete overview of Pippenger's algorithm with a detailed explanation of the different steps. To end with this part, we give a pseudocode of Pippenger's algorithm in Subsection 3.2.4 followed by a calculation of its computational complexity and, finally a description of the signed version of Pippenger's algorithm. In Section 4, we propose our optimization of Pippenger's algorithm designed for memory-constrained devices, and then expose the results we obtained in comparison to Pippenger's algorithm in Subsection 4.2.

2 Background

2.1 Elliptic curves

In this section, we present the elliptic curve operations that are relevant for efficient implementations. For a more detailed explanation, the reader may refer to [17].

Depending on cryptographic protocols, various types of elliptic curves and coordinate systems are used. A point in affine coordinates can be represented with two field elements (x, y) that satisfy an elliptic curve equation $y^2 = x^3 + ax + b$. To improve efficiency, many coordinate systems have been invented. For example, projective coordinates are represented by three coordinates X, Y, Z with $x = X/Z$, $y = Y/Z$. Projective coordinates are very common and often used in this type of operations. We have therefore chosen to perform our tests using them. However, there are many coordinate systems, some allow operations to be performed more efficiently but often require more memory. If we refer to the Explicit-Formulas Database [13], we can see that elliptic curves in twisted Edward form with extended coordinates (with $a = 1$)

[10] have the lowest costs for elliptic curve operations. This form of elliptic curve is thus preferred for computations of large MSMs [14]. For a detailed list of the different curves and coordinates, we refer to the Explicit-Formulas Database [13]. If P, Q are two points in the same coordinate system with $P \neq Q$, we will denote **ADD** the operation $P + Q$ and **DBL** the operation $2P$. On the other hand, if P, Q are two points, one in affine coordinates and the other in a different coordinate system, with $P \neq Q$, we will denote **MADD** (for Mixed ADDition) the operation $P + Q$.

For testing and comparisons, we take the example of BLS12-381 as it is a widely used curve in cryptographic protocols, especially in zk-SNARKs protocols. BLS12-381 [15] is the curve defined by $E : y^2 = x^3 + 4$. To support pairings operations, this elliptic curve is defined over two fields : the base field $\mathbb{F}_p$ and the extension field $\mathbb{F}_{p^2}$. However, in this paper, we only consider the curve defined over $\mathbb{F}_p$ (with $p = 2^{381} - 19$), since MSM is performed in the subgroup $\mathbb{G} \subset E(\mathbb{F}_p)$. This elliptic curve belongs to the BLS (Barreto-Lynn-Scott) family of curves that are all pairing-friendly [5]. The group $E(\mathbb{F}_p)$ has order $q \times r$ and in the following $\mathbb{G}$ will constantly refer to the subgroup of order q (which is a 255-bit prime) in $E(\mathbb{F}_p)$. As we said in the previous paragraph, we use projective coordinates to represent the resulting points (that are the result of an operation). The corresponding costs for the curve operations performed in this coordinate system, assuming that one modular squaring (**S**) is equivalent to 0.8 modular multiplication (**M**), are: 1**ADD** = 12**M**, 1**MADD** = 10.6**M** and 1**DBL** = 7**M**.

2.2 Multi-Scalar Multiplication

Multi-Scalar Multiplication is one of the fundamental operations in elliptic curve cryptography. It is based on calculating the sum of several multiplications of elliptic curve points by scalars. Given $\mathbb{G} \subseteq E(\mathbb{F}_p)$, a subgroup of order q (with bit size b), consider $P_1, \ldots, P_n \in \mathbb{G}$ as n points, and $a_1, \ldots, a_n \in \mathbb{F}_q$ as n scalars, the goal is to compute:

$$R = \sum_{i=1}^{n} a_i P_i = a_1 P_1 + a_2 P_2 + \cdots + a_n P_n$$

What will primarily concern us in this paper is the use of MSM when n is large. The most common method to calculate MSM efficiently is Pippenger's algorithm.

3 Multi-Scalar Multiplication algorithms

3.1 Straus's algorithm

Straus's method [31], also known as Shamir's trick, is one of the earliest algorithms for Multi-Scalar Multiplication. It only requires $O(1)$ memory [4] during the execution of the algorithm, unlike Pippenger's algorithm which requires $O(2^w)$. However, we have to precompute some points (precisely 2^{nw} points), which is not necessary in Pippenger's algorithm. That is why this method is not suitable for large n, because the precomputation part would be enormous.

First, we choose a small integer window w (as in Pippenger's algorithm) and divide each scalar a_i (in its binary form) into $\lceil \frac{b}{w} \rceil$ parts $a_{i,j}$ with $j = 1, \ldots, \lceil \frac{b}{w} \rceil$. Then, we

precompute all the possible linear combinations and create a table T:

$$T = \left\{ b_1 P_1 + b_2 P_2 + \cdots + b_n P_n \;\middle|\; 0 \leq b_i \leq 2^w - 1 \text{ with } i = 1, \cdots, n \right\}$$

Remark 1 As we said before, this precomputation table T (with 2^{nw} points) is exponentially large and quickly becomes impossible to store as the size of n increases.

At the j-th step of the algorithm, the accumulator R is multiplied by 2^w, and the precomputed value $\sum_{i=1}^{n} a_{i,j} P_i$ is added directly to R. Since additions are performed immediately, no buckets are stored. This method is efficient for small MSMs (e.g. signature verification), but it is outperformed by Pippenger's algorithm for large-scale computations.

Algorithm 1 Straus's Multi-Scalar Multiplication

Require: A scalar vector $a = (a_1, \ldots, a_n)$ and the precomputed table T
Ensure: $R = \sum_{i=1}^{n} a_i P_i$
1: $R \leftarrow 0_E$ ▷ Initialize R
2: **for** $j = \lceil b/w \rceil - 1$ *down to* 0 **do**
3: $R \leftarrow 2^w R$ ▷ Perform w doublings
4: $R \leftarrow R + T_j$ ▷ T_j is the precomputed linear combination of the points corresponding to the w bits of the window at iteration j
5: **end for**
6: **return** R

There exists a variant of Straus's algorithm which requires a smaller precomputation table (with only $n(2^w - 1)$ points):

$$P = \left\{ b_i P_i \;\middle|\; 1 \leq b_i \leq 2^w - 1 \text{ with } i = 1, \cdots, n \right\}$$

At j-th step, we add together the precomputed points $a_{i,j} P_i$ for $i = 1, \cdots, n$ to obtain $\sum_{i=1}^{n} a_{i,j} P_i$ that we add to R. This variant saves some storage space but we will see in the following part that more recent methods such as Pippenger's algorithm scale better for large MSMs.

3.2 Pippenger's algorithm

The bucket method [26] invented by Pippenger in 1976 is the most efficient technique to calculate the MSM, particularly when dealing with a large number of points. The algorithm we propose in Section 4.1 is strongly inspired by Pippenger's algorithm. Our exposition of Pippenger's algorithm follows the description given by Botrel and El Housni in [14], which provides a clear presentation. The core idea of Pippenger's algorithm is to decompose the MSM into smaller, more easily computable MSMs and

then combine the results from these smaller MSMs to obtain the final result. Now, let us see how to put this into practice.

3.2.1 Step 1: Reduce the MSM into several smaller MSMs

First, we choose a window $w \leq b$ and write each scalar a_i in binary form. Then, we divide each a_i into $\lceil \frac{b}{w} \rceil$ parts $a_{i,j}$ with $j = 1, \ldots, \lceil \frac{b}{w} \rceil$. In the second part of this method, we will use these $a_{i,j}$ as bucket indexes. From this decomposition, we can construct $\lceil \frac{b}{w} \rceil$ smaller MSMs M_j, for $j \in [\![1, \ldots, \lceil \frac{b}{w} \rceil]\!]$.

$$\forall j \in [\![1, \ldots, \lceil b/w \rceil]\!], M_j = \sum_{i=1}^{n} a_{i,j} P_i. \tag{1}$$

Example 1 Let P_1, P_2, P_3 three points in $\mathbb{G}$, a_1, a_2, a_3 three scalars which are respectively equals to $187, 201, 138$.

- We choose $w = 3 < b = 8$ as a window.
- We write each scalar $a_1 = 187, a_2 = 201, a_3 = 138$ in binary form and partition each into $\lceil \frac{8}{3} \rceil = 3$ parts of length w.

$$a_1 = (\underbrace{10}_{2}, \underbrace{111}_{7}, \underbrace{011}_{3}), a_2 = (\underbrace{11}_{3}, \underbrace{001}_{1}, \underbrace{001}_{1}), a_3 = (\underbrace{10}_{2}, \underbrace{001}_{1}, \underbrace{010}_{2})$$

$$a_1 = (2, 7, 3), a_2 = (3, 1, 1), a_3 = (2, 1, 2)$$

- We construct $\lceil \frac{b}{w} \rceil = \lceil \frac{8}{3} \rceil = 3$ smaller MSMs M_j from the partitioned scalars:

$$M_1 = 2P_1 + 3P_2 + 2P_3$$

$$M_2 = 7P_1 + 1P_2 + 1P_3$$
$$M_3 = 3P_1 + 1P_2 + 2P_3$$

3.2.2 Step 2: Efficiently solve these smaller MSMs

Now, we solve these small MSMs by grouping the points into $2^w - 1$ buckets (initialized with points at infinity) for each M_j. To do so, for each $i \in [\![1, n]\!]$, we just accumulate the point P_i in bucket number $a_{i,j}$. Then, once all the buckets are filled, for each M_j, we obtain $2^w - 1$ sums $S_1, \ldots, S_{2^w-1}$ by adding the points that are in the same bucket, and we just have to compute $S_1 + 2S_2 + 3S_3 + \cdots + (2^w - 1)\, S_{2^w-1}$ to get the value of M_j. As the scalars are ordered, we can calculate this sum efficiently using only $2(2^w - 2)$ additions.

$$S_{2^w-1} + (S_{2^w-1} + S_{2^w-2}) + \cdots + (S_{2^w-1} + S_{2^w-2} + \cdots + S_2 + S_1)$$

Example 2 Always with $w = 3$, let's take an arbitrary example with $M_j = 2P_1 + 6P_2 + P_3 + 7P_4 + 6P_5 + 2P_6 + 6P_7 + 3P_8 + 4P_9 + 5P_{10} + P_{11} + 3P_{12} + 3P_{13}$.

- Initialise $2^w - 1 = 7$ empty buckets with points at infinity and accumulate the points P_i into the bucket corresponding to the scalar which multiplies P_i.

			P_{13}			P_7	
			$+$			$+$	
	P_{11}	P_6	P_{12}			P_5	
	$+$	$+$	$+$			$+$	
	P_3	P_1	P_8	P_9	P_{10}	P_2	P_4
buckets:	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	$\boxed{7}$
Sum:	S_1	S_2	S_3	S_4	S_5	S_6	S_7

- Combine the buckets to compute $M_j = 2P_1 + 6P_2 + P_3 + \cdots$:

$$
\begin{array}{llllllllllll}
 & S_7 & & & & & & & & & & \\
+ & S_7 & + & S_6 & & & & & & & & \\
 & \vdots & & \vdots & & & & & & & & \\
+ & S_7 & + & S_6 & + & \cdots & + & S_3 & + & S_2 & & \\
+ & S_7 & + & S_6 & + & \cdots & + & S_3 & + & S_2 & + & S_1 \\
\hline
M_j = & \underbrace{7S_7}_{7P_4} & + & \underbrace{6S_6}_{6P_2+6P_5+6P_7} & + & \cdots & + & \underbrace{3S_3}_{3P_8+3P_{12}+3P_{13}} & + & \underbrace{2S_2}_{2P_1+2P_6} & + & \underbrace{S_1}_{P_3+P_{11}}
\end{array}
$$

With this method, we obtain $M_j = 2P_1 + 6P_2 + P_3 + \cdots$ using only two additions per line as we reuse the result of the previous line at each step.

3.2.3 Step 3: Combine the results to obtain the final outcome

This step is the classical way of computing an exponentiation. The following algorithm combines the small MSMs into the original one.

$$\sum_{i=1}^{n} a_i P_i = 2^w(\cdots(2^w(2^w M_1 + M_2) + M_3)\cdots + M_{\lceil b/w \rceil} \tag{2}$$

3.2.4 Pseudo-code of Pippenger's algorithm

Algorithm 2 presents the naive way to implement Pippenger's algorithm as presented in [32] because it is easier to differentiate the different steps of the method, but in practice, most people use Algorithm 3 where we only generate one set of $2^w - 1$ buckets [23, 1, 22].

Algorithm 2 Naive Pippenger's algorithm

Require: A scalar vector $a = (a_1, \ldots, a_n)$, a point vector $P = (P_1, \ldots, P_n)$
Ensure: $R = a_1 P_1 + \cdots + a_n P_n$

Step 1
1: Decompose each scalar $a_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,\lceil b/w \rceil})$

Step 2
2: $S_{j,k} \leftarrow 0_E$, $\forall j \in [\![1, \lceil b/w \rceil]\!]$, $\forall k \in [\![1, 2^w - 1]\!]$ ▷ Initialize the buckets
3: **for** $i = 1$ *to* n **do**
4: **for** $j = 1$ *to* $\lceil \frac{b}{w} \rceil$ **do**
5: **if** $a_{i,j} \neq 0$ **then** ▷ $a_{i,j}$ is the j-th chunk of scalar a_i
6: $S_{j,a_{i,j}} \leftarrow S_{j,a_{i,j}} + P_i$ ▷ Accumulate the points in the buckets
7: **end if**
8: **end for**
9: **end for**
10: $M_j \leftarrow 0_E$ $\forall j \in [\![1, \lceil b/w \rceil]\!]$
11: **for** $j = 1$ *to* $\lceil \frac{b}{w} \rceil$ **do**
12: $T \leftarrow 0_E$
13: **for** $k = 2^w - 1$ *down to* 1 **do** ▷ Combine the buckets to compute the M_j
14: $T \leftarrow T + S_{j,k}$ ▷ T corresponds to one row in the addition shown in example 2
15: $M_j \leftarrow M_j + T$
16: **end for**
17: **end for**

Step 3
18: $R = M_1$
19: **for** $j = 2$ *to* $\lceil b/w \rceil$ **do** ▷ Calculate the outcome by combining the M_j
20: $R \leftarrow 2^w R + M_j$
21: **end for**
22: **return** R
23:

Algorithm 3 Preferred implementation of Pippenger's algorithm

Require: $a = (a_1, \ldots, a_n)$, $P = (P_1, \ldots, P_n)$, the window w
Ensure: $R = a_1P_1 + \cdots + a_nP_n$

$R \leftarrow 0_E$
for $j = 1$ *to* $\lceil b/w \rceil$ **do**
  $R \leftarrow 2^w R$
  $S_r \leftarrow 0_E$, $\forall r \in [\![1, 2^w - 1]\!]$
  for $i = 1$ *to* n **do**
    $S_{a_{i,j}} \leftarrow S_{a_{i,j}} + P_i$ ▷ Add P_i in the bucket number $a_{i,j}$
  end for
  $T \leftarrow 0_E$
  for $m = 2^w - 1$ *down to* 1 **do**
    $T \leftarrow T + S_m$
    $R \leftarrow R + T$
  end for
end for
return R

3.2.5 Computational complexity of Pippenger's algorithm

- **step 1**: Negligible.
- **step 2**: For one T_j, we use $n - (2^w - 1)$ **MADD** to compute the S_i and $2(2^w - 2)$ **ADD** to compute T_j from the S_i. As we have $\lceil \frac{b}{w} \rceil$ T_j, step 2 requires in total $2(2^w - 2) \lceil \frac{b}{w} \rceil$ **ADD** and $(n - (2^w - 1)) \lceil \frac{b}{w} \rceil$ **MADD**.
- **step 3**: $\lceil \frac{b}{w} \rceil - 1$ **ADD** and $\left(\lceil \frac{b}{w} \rceil - 1 \right) w$ **DBL** to calculate the final outcome.

Total: $\lceil \frac{b}{w} \rceil \left(2^{w+1} - 3 \right) - 1$ **ADD**, $\lceil \frac{b}{w} \rceil (n - (2^w - 1))$ **MADD**, $\left(\lceil \frac{b}{w} \rceil - 1 \right) w$ **DBL**.

3.2.6 Signed-digit version of Pippenger's algorithm

An interesting optimization of Pippenger's algorithm is the signed-digit version. Instead of using the classical binary form for the scalar representation, we use a Non-Adjacent Form (NAF) decomposition. If we write the scalars $a_{i,j}$ in the set $[\![-2^{w-1}, 2^{w-1} - 1]\!]$, we can reduce the number of buckets by half doing that because computing $-P_i$ is free: $\begin{cases} \text{If } a_{i,j} > 0, \text{ we add } P_i \text{ in the bucket } S_{a_{i,j}} \\ \text{If } a_{i,j} \leq 0, \text{ we add } -P_i \text{ in the bucket } S_{|a_{i,j}|} \end{cases}$

Since we have reduced the number of buckets by half, we now have fewer additions $\left(\lceil \frac{b}{w} \rceil (2^w - 3) - 1 \right)$ and more mixed additions $\left(\lceil \frac{b}{w} \rceil \left(n - \left(2^{w-1} - 1 \right) \right) \right)$, which are much less costly, thus reducing the overall complexity of the algorithm. Furthermore, by reducing the number of buckets by half, we also save a lot of memory.

4 Optimization of MSM on memory-constrained devices

In cryptographic computations, especially when dealing with zk-SNARKs, memory management plays a crucial role. Pippenger's algorithm can require a large amount of memory when working with a large number of points. Thus, efficient memory usage is often a necessity when zk-SNARKs are implemented on devices with limited storage, such as mobile devices or embedded systems. Given that Pippenger's algorithm involves the accumulation of points in various buckets during the MSM process, the amount of memory required can grow significantly with the size of n. This can lead to bottlenecks or even make it infeasible to run such operations on memory-constrained devices.

First, we need to store the points $P_1, \ldots, P_n \in \mathbb{G}$ and the n scalars $a_1, \ldots, a_n \in \mathbb{F}_q$. Furthermore, each point is stored in decompressed affine coordinates, so it is represented by two elements of $\mathbb{F}_p$, and each scalar is an element of $\mathbb{F}_q$. As we said in Subsection 2.1, we work on BLS12-381, then to store a single element of $\mathbb{F}_p$, 381 bits are required, which equals 48 bytes. Therefore, to store a single point in $\mathbb{G}$, 96 bytes are needed. For a scalar in $\mathbb{F}_q$, 255 bits are needed, which equals 32 bytes. In summary, whether it is our proposed algorithm 4 or Pippenger's algorithm, $128 \times n$ bytes are required to store the points and scalars. For clarity, we assume that these elements are stored externally, and the results we present account only for the memory required by the algorithm itself.

All operation results such as the sums S_r in the buckets are in projective coordinates, so we need 3 field elements to represent them. If we want to optimize the computational complexity, we could use another representation system, such as extended coordinates (on twisted Edwards curves) that have a lowest cost for the operations on the curve, but it would increase the memory usage (4 field elements instead of 3). As we said in section 1.2, we also tested our algorithm with extended coordinates on twisted Edwards curves and obtained very similar results.

In Section 4.1, we propose a variant of Pippenger's algorithm to optimize MSM on devices with limited memory. We also created a signed version of this algorithm which works exactly the same way but is, of course, more efficient. We omit the details here, since the adaptation to the signed method is similar to Subsection 3.2.6. First, let us examine the storage requirements for Pippenger's algorithm. They need to store $2^w - 1$ resulting points (the sums in the buckets) and two others for T and R. Moreover, they need 3 field elements to represent one point in projective coordinates, so 144 bytes to store one resulting point. In total, they have to use $144 \times (2^w + 1)$ bytes of memory to perform an MSM with n points of BLS12-381. As n increases, the optimal window size w also grows, leading to higher memory consumption. This is exactly what we want to avoid with our variant.

4.1 Balanced variant with an adapted number of buckets

We present an algorithm inspired by Algorithm 3 that works faster than Pippenger's algorithm when it cannot choose the optimal window because of memory limitations. When we refer to the optimal window w, we mean the choice of w that minimizes the

Table 1 Evolution of $w_{optimal}$ in function of n in Pippenger's algorithm for computing an MSM on BLS12-381 with points in projective coordinates

n	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
$\mathbf{w_{optimal}}$	2	3	3	4	5	6	7	8	8	9
n	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
$\mathbf{w_{optimal}}$	10	11	12	13	13	15	15	15	17	17

computational complexity of Pippenger's algorithm, assuming no memory limitations. This optimal window is chosen by testing the different window values and selecting the most efficient one. We can approximate the value of $w_{optimal}$ by calculating it using the computational complexity formula, but it may differ from the real value. However, we can use this approximation to only test 3 values (calculated $w_{optimal} \pm 1$) and this is actually, the best way to select the right $w_{optimal}$ [22, 23]. This step of finding the optimal window size can be performed offline before computing the MSM. We give the $w_{optimal}$ values depending on n obtained in our tests in Table 1.

In our algorithm, instead of using $2^w - 1$ buckets, we will choose a specific number of buckets d, depending on the available memory (we choose d as large as we can). For a fixed memory, d can be calculated as follows:

$$d = \left\lfloor \frac{\text{Available memory in bytes}}{\text{Memory to store one resulting point}} \right\rfloor - 2$$

In $\text{BLS12} - 381$, the necessary memory to store one bucket is equal to $3 \times 48 = 144$ bytes, so the formula is becomes: $d = \lfloor \frac{\text{Available memory in bytes}}{144} \rfloor - 2$. It is exactly the maximum number of buckets we can store minus 2 because we have to keep some memory for T and R. Thus, instead of initializing $2^w - 1$ buckets for each instance M_j, we will now use d buckets to accumulate the points. First, we accumulate the points P_i that are multiplied by $2^w - 1 - d < a_i \leq 2^w - 1$ in the d buckets (corresponding to the buckets $B_{2^w-1-d}, \ldots, B_{2^w-1}$) and calculate the sums S_j. Then we add the sums S_j to T and add T to R (see Algorithm 4). Finally, we empty the buckets and reuse them to accumulate the points that are multiplied by $2^w - 1 - 2d < a_i \leq 2^w - 1 - d$, calculate the corresponding sums S_j etc. We repeat this process until we have used all the points.

The evolution of parameter d in AdaptiveSetBucket is similar to that of $2^w - 1$ in Pippenger's algorithm. Indeed, until d reaches $2^{w_{optimal}} - 1$, the increasing of d leads to a reduction of the number of CPU cycles. We can observe this phenomenon on Figure 2 (the evolution of memory corresponds to the evolution of d). Starting from $2^{w_{optimal}} - 1$, increasing d no longer really has an effect and will not fundamentally change the complexity. Therefore, it makes sense to fix d to $2^{w_{optimal}} - 1$, as changing it will not have a significant impact.

Algorithm 4 AdaptiveSetBucket

Require: $a = (a_1, \ldots, a_n)$, $P = (P_1, \ldots, P_n)$, the window w, the number of buckets d
Ensure: $R = a_1P_1 + \cdots + a_nP_n$

```
1:  R ← 0_E
2:  for j = 1 to ⌈b/w⌉ do
3:      R ← 2^w R
4:      T ← 0_E
5:      for k_1 = 2^w − 1 down to 1 with step −d do
6:          k_0 ← max(1, k_1 − d + 1)
7:          S_r ← 0_E, ∀r ∈ [[1, d]]          ▷ Initialize d buckets instead of 2^w − 1
8:          for i = 1 to n do
9:              if k_0 ≤ a_{i,j} ≤ k_1 then
10:                 S_{a_{i,j}−k_0} ← S_{a_{i,j}−k_0} + P_i
11:             end if
12:         end for
13:         for i = k_1 to k_0 with step −1 do
14:             T ← T + S_{i−k_0}
15:             R ← R + T
16:         end for
17:     end for
18: end for
19: return R
```

Using this method, we need to store $d + 2$ resulting points (d points for the d buckets) and two others for T and R. Furthermore, we can always select the best window for our algorithm because the window value is independent from the number of buckets (and thus, not impacted by memory limitations). Let see in Example 3 why we have better results using our algorithm instead of Pippenger's one, on a limited memory device.

Example 3 In this example, the goal is to compute an MSM on BLS12-381 with 2^{13} points on a device with a small memory of 15 KB. Normally, the optimal window size for this number of points is $w = 10$ (verified experimentally), but in this case, it is not possible to use this window because it would require storing $2^{10} - 1$ buckets which consumes approximately 147.32 KB $>$ 15 KB. $w = 9$, $w = 8$ and $w = 7$ do not work for the same reason. Thus, it is necessary to use $w = 6$ (To store $2^6 - 1$ buckets, this requires approximately 9.08 KB $<$ 15 KB) to perform this MSM. On the other hand, our algorithm do not have to limit its choice to $2^6 - 1 = 63$ buckets, it will choose the highest d possible, which is $d = 104$ buckets in this case (it requires 14.98 KB $<$ 15 KB). By doing that, it optimizes the available storage and reduces its computational complexity as we can see in the line 2 of Table 2.

AdaptiveSetBucket has nearly the same computational complexity formula than Pippenger's algorithm, except that $2^w - 1$ is replaced by d and the window $w_{adaptive}$ is always greater or equal than $w_{Pippenger}$. The only thing that could be a problem is that we do many more comparisons than in Pippenger's algorithm. Specifically, we

make $2n \times \lceil b/w \rceil \times \lceil (2^w - 1)/d \rceil$ comparisons. However, we can observe experimentally in the next part that it is not very important in comparison to our gain.

4.2 Experimental results

Our algorithm is implemented in the C programming language, and all experiments were performed on a Dell Inc. OptiPlex 7450 AIO running Ubuntu 24.04.3 LTS. It is equipped with an Intel Core $i5\text{-}7500 \times 4$ CPU and 16 GB of RAM. We use the baseline implementation provided by the RELIC toolkit [2], to compare Algorithm 3 with Algorithm 4. For comparisons, we first fix the number of MSM points and generate them randomly. For Pippenger's algorithm, we execute the procedure 30 times for each window size $2 < w \leq w_{optimal}$, using freshly generated random scalars in every iteration. We then calculate the average number of CPU cycles per window size. For the AdaptiveSetBucket algorithm, we follow a similar methodology: the algorithm is executed 30 times for each bucket size, with the same randomly generated scalars than in Pippenger's algorithm. The average CPU cycle count is also recorded. The number of buckets is gradually increased, starting from 4. Specifically, we increase by 1 up to 256, by 2 up to 512, by 3 up to 1024, etc., until we reach $2^{w_{optimal}+1}$, where $w_{optimal}$ denotes the optimal window size for AdaptiveSetBucket. Once the memory is not a limitation, we could also take the same parameters than in Pippenger's algorithm and it would work exactly the same way.

Each set of experiments is repeated twice for both algorithms and, in order to reduce fluctuations due to machine-level variability, we report the minimum values obtained across the two runs. The comparison graphs below presents the results for MSMs with 2^9 and 2^{13} input points, but we have similar results for MSMs with $2^{10}, 2^{11}, 2^{12}$ and 2^{14} points.

Remark 2 We also compared the two algorithms on an MSM of size 2^{18} points and obtained similar results. But instead of 30 times, we ran AdaptiveSetBucket 5 times for each execution, and we tried only 28 different numbers of buckets (2 for each window in Pippenger's algorithm, from 2 to 15. 15 being the optimal window to calculate an MSM of 2^{18} points). Both algorithms perform equally well when memory is not limited. In contrast, if we consider a memory limited to 32KB, our algorithm performs 10% faster.

As we can see in the following figures, when sufficient memory is available for Pippenger's algorithm to select its optimal window, our approach achieves comparable performance. Nevertheless, under memory constraints, our method clearly outperforms Pippenger's algorithm.

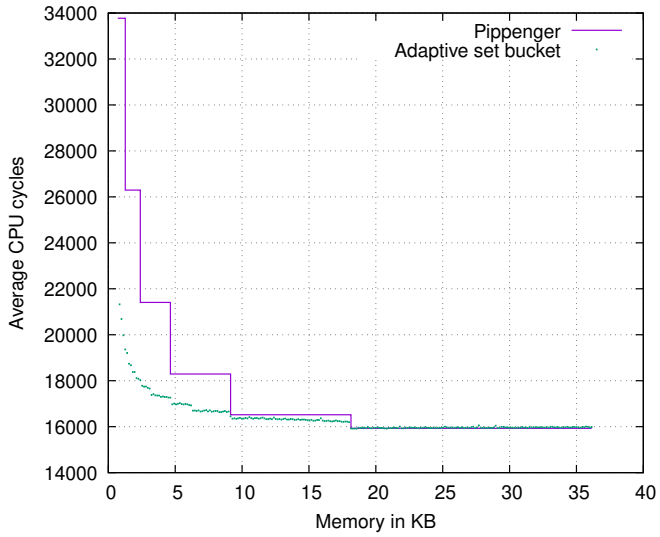

Fig. 1 Complexity comparison between Pippenger's algorithm and AdaptiveSetBucket on the calculation of an MSM with 2^9 points, depending on the available memory

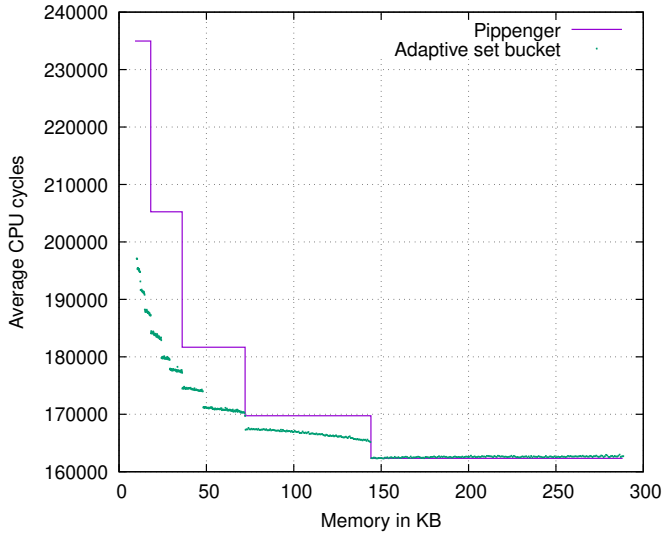

Fig. 2 Complexity comparison between Pippenger's algorithm and AdaptiveSetBucket on the calculation of an MSM with 2^{13} points, depending on the available memory

By looking at Table 1, we can observe that by increasing the number of points n in the MSM, the size of the optimal window increases too. Moreover, the number of operations increases drastically when the size of n increases (n being the most important number in the complexity formula). If we now look at Figures 1 and 2 , we can observe that for a fixed number of points in the MSM, as memory increases, the

number of operations required for the operation of Pippenger and AdaptiveSetBucket decreases, until it reaches its minimum when memory is no longer a constraint for choosing the window w.

For instance, considering an MSM with 2^{13} points, we can observe on Figure 2 that between ≈ 144 KB and ≈ 290 KB, AdaptiveSetBucket and Pippenger's algorithm have a very similar efficiency because they both use the same window w and approximately the same number of buckets. Indeed, the optimal window for Pippenger's algorithm in this case is $w = 10$. It corresponds to a required memory of at least 147.6KB ($144 \times (2^{10} + 1)$ bytes). Thus, from 147.6KB, both algorithm will use approximately the same number of buckets (the optimal one). On the other side, if the memory is limited, for instance, between 19KB and 35KB, Pippenger's algorithm selects $w = 7$ and thus uses $2^7 - 1 = 127$ buckets (it cannot choose a higher window because it would require too much memory for the buckets. For example $w = 8$ would require $144 \times (2^8 + 1)$ bytes ≈ 37.01KB.). On the other hand, AdaptiveSetBucket has a more subtle approach and can adapt its number of buckets, from 127 buckets (which require ≈ 18.29KB) to 246 buckets (which require ≈ 35.28KB) depending on the available memory. More specifically, if we now use a device with a memory of 15 KB, we have a gain of 20% compared to Pippenger's algorithm thanks to the adaptation of the number of buckets and the fact that we can select a higher window without increasing the required storage.

Table 2 Parameters, memory and average CPU cycles for an MSM with 2^{13} points

Memory in KB	**Algorithm**	**w**	**Number of buckets d**	**Average CPU cycles**	**Percentage gain**
9	Pippenger AdaptiveSetBucket	5 8	$2^5 - 1 = 31$ 61	274039 200871	$\approx 26.70\%$
15	Pippenger AdaptiveSetBucket	6 9	$2^6 - 1 = 63$ 104	234968 188185	$\approx 19.91\%$
20	Pippenger AdaptiveSetBucket	7 9	$2^7 - 1 = 127$ 140	205245 184000	$\approx 10.35\%$
35	Pippenger AdaptiveSetBucket	7 10	$2^7 - 1 = 127$ 246	205245 177541	$\approx 13.50\%$
50	Pippenger AdaptiveSetBucket	8 10	$2^8 - 1 = 255$ 353	181672 171115	$\approx 5.81\%$
70	Pippenger AdaptiveSetBucket	8 10	$2^8 - 1 = 256$ 495	181672 170468	$\approx 6.17\%$
100	Pippenger AdaptiveSetBucket	9 10	$2^9 - 1 = 511$ 709	169744 166913	$\approx 1.67\%$
140	Pippenger AdaptiveSetBucket	9 10	$2^9 - 1 = 511$ 993	169744 165584	$\approx 2.45\%$
175	Pippenger AdaptiveSetBucket	10 10	$2^{10} - 1 = 1023$ 1023	162334 162359	$\approx -0.02\%$

In Table 2, we present some results obtained while computing an MSM of size 2^{13}. For a fixed memory (represented in Kilobytes), we compare the efficiency of Pippenger's algorithm with our AdaptiveSetBucket algorithm by counting the average number of CPU cycles in each algorithm over 30 iterations. We also show the number of buckets used in each algorithm and which window w is chosen. We can notice that the window in AdaptiveSetBucket is always greater or equal than Pippenger's one. As we can see in the last row, for a fixed memory of 175 KB, Pippenger's algorithm is no longer limited by the memory to select its optimal window. In this case, AdaptiveSetBucket could be a bit less efficient than Pippenger's algorithm for some tests. But it depends, sometimes we have better values than Pippenger's algorithm, even if the memory is no longer a constraint. These small differences depend on the tests, and are never greater than 1%. If we want to avoid them, we just have to fix the number of buckets to $2^{w_{optimal}}$ when memory is no longer a constraint. By doing that, AdaptiveSetBucket works exactly the same way as Pippenger's algorithm, and has the same complexity.

5 Conclusion

In this paper, we introduced AdaptiveSetBucket, an efficient variant of Pippenger's algorithm for Multi-scalar Multiplication tailored for memory-constrained devices. The central idea is to choose the number of buckets d based on available memory, instead of the fixed $2^w - 1$ buckets used in Pippenger's algorithm. This algorithm leads to better utilization of the memory footprint and, thus, higher overall performance.

The next step would be to try to use this algorithm to compute large MSMs on devices with limited memory, for example during Groth16 proof generation. Furthermore it could be interesting to test our algorithm with parallelization methods because it seems to be totally compatible with this kind of optimization.

References

[1] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. FPGA acceleration of multi-scalar multiplication: CycloneMSM. Cryptology ePrint Archive, Paper 2022/1396, 2022. https://eprint.iacr.org/2022/1396.

[2] Diego F. Aranha and contributors. Relic toolkit.

[3] Michael Clear Aritra Banerjee and Hitesh Tewari. Demystifying the role of zk-snarks in zcash. 2020 IEEE Conference on Application, Information and Network Security (AINS), 2020. https://ieeexplore.ieee.org/document/9315064.

[4] Roberto M. Avanzi. The complexity of certain multi-exponentiation techniques in cryptography. *Journal of Cryptology*, 18(4):357–373, 2005.

[5] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In S. Cimato, G. Persiano, and C. Galdi, editors, *Security in Communication Networks (SCN) 2002*, volume 2576 of *Lecture Notes in Computer Science*, pages 257–267. Springer, Berlin, Heidelberg, 2003.

[6] 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, May 2014. ISSN: 2375-1207.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8043, pages 90–108. Springer Berlin Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 90–108. Springer, 2013.

[9] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023, pages 389–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[10] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, Berlin, Heidelberg, 2008.

[11] 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.

[12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[13] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. https://hyperelliptic.org/EFD/index.html, 2007.

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

[15] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (S&P)*, pages 1080–1097. IEEE, 2020.

[16] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. A review of zk-snarks, 2023.

[17] Alfred Menezes Darrel Hankerson, Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. Springer-Verlag, New York, 2004.

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

[19] gnark. Modular multiplication, 2023. Accessed: 2024-08-21.

[20] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances*

in Cryptology – EUROCRYPT 2016, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, Heidelberg, 2016.

[21] Youssef El Housni and Aurore Guillevic. Families of snark-friendly 2-chains of elliptic curves. In Olivier Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, volume 13276 of *Lecture Notes in Computer Science*, pages 367–396. Springer, Cham, 2022.

[22] Rui Jiang, Cong Peng, Min Luo, Rongmao Chen, and Debiao He. Simdmsm: Simd-accelerated multi-scalar multiplication framework for zksnarks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025:681–704, 03 2025.

[23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):194–220, 2023.

[24] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. IACR Transactions on Cryptographic Hardware and Embedded Systems, vol 2023, Issue 2, pages 358-380, 2023.

[25] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, May 2013. ISSN: 1081-6011.

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

[27] Xander Pottier, Thomas de Ruijter, Jonas Bertels, Wouter Legiest, Michiel Van Beirendonck, and Ingrid Verbauwhede. OPTIMSM: FPGA hardware accelerator for Zero-Knowledge MSM, 2024. Publication info: Published by the IACR in TCHES 2025.

[28] Xavier Salleras and Vanesa Daza. ZPiE: Zero-Knowledge Proofs in Embedded Systems. *Mathematics*, 9(20):2569, January 2021.

[29] Runnan Shen, Liang Wang, Haotian Luo, Rui Yang, Jinqian Yang, Jinquan Wang, Qiancheng Sun, Limin Xiao, and Jin Dong. Accelerating zk-snark with group and zone optimization on gpu. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 24–31, 2023.

[30] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 2 edition, 2009.

[31] E. G. Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70:806–808, 1963.

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