Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs

Anonymous Submission

Abstract. The arithmetic of computing multiple scalar multiplications in an elliptic curve group then adding them together is called multi-scalar multiplication (MSM). MSM over fixed points dominates the time consumption in the pairing-based trusted setup zero-knowledge succinct non-interactive argument of knowledge (zkSNARK), thus for practical applications we would appreciate fast algorithms to compute it. This paper proposes a bucket set construction that can be utilized in the context of Pippenger's bucket method to speed up MSM over fixed points with the help of precomputation. If instantiating the proposed construction over BLS12-381 curve, when computing n-scalar multiplications for $n = 2^e$ $(10 \leq e \leq 21)$, theoretical analysis indicates that the proposed construction saves more than 21% computational cost compared to Pippenger's bucket method, and that it saves 2.6% to 9.6% computational cost compared to the most popular variant of Pippenger's bucket method. Finally, our experimental result demonstrates the feasibility of accelerating the computation of MSM over fixed points using large precomputation tables as well as the effectiveness of our new construction.

Keywords: Multi-scalar multiplication · Pippenger's bucket method · zkSNARK · blockchain

1 Introduction

In recent years, zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) has gained tremendous interest from its theoretical development to the practical implementation because it provides an elegant privacy protection solution. Popular examples include anonymous transactions in Zcash [BCG$^+$14] and smart contract verification over private inputs [Ebe] in Ethereum. Many zkSNARKs with trusted setup rely on pairing-based cryptography, which are very efficient in general. Groth et al. [GOS06, Gro06, Gro09, Gro10, GOS12, GS12] first introduced pairing-based zero-knowledge proofs, leading to the extensive research work in this area [Lip12, GGPR13, DFGK14, Gro16, MBKM19, GWC19, CHM$^+$19, BFS20, BDFG21].

All pairing-based trusted setup zkSNARKs in the literature follow a common paradigm, where the prover computes a proof consisting of several elements in an elliptic curve group by generic group operations and the verifier checks the proof by a number of pairings in the verification equation. Basically, it requires the prover and verifier to conduct their computation by only using linear operations to the elliptic curve points built in the common reference string. This computation is indeed MSM over fixed points. MSM dominates the overall time for generating and verifying the proof. Thus, fast algorithms for computing MSM over fixed points are desirable and necessary.

MSM in those applications shows the characteristic of having a large amount of points. For example, one of the most classical zkSNARK applications is to prove the knowledge of preimage for a cryptographic hash function. When using the traditional SHA-256, which is compiled to an arithmetic circuit with 22,272 AND gates when the preimage is 512 bit [CGGN17], it will lead to the computation of MSM with more than 22,272 points.

When utilizing zkSNARK-friendly hash function Poseidon [GKR+21], the MSM still has hundreds of points.

1.1 Related work

The most popular method for scalar multiplication in the elliptic curve group is the binary algorithm, known as doubling and addition method (also known as square and multiplication method in the exponentiation setting) [Knu97, Section 4.6.3]. GLV method [GLV01] and GLS method [GLS11] decompose the scalar into dimensions 2,4,6 and 8, then compute the corresponding MSM. When the point for scalar multiplication is fixed, precomputation can be used to reduce the computational cost. Knuth's 5 window algorithm utilizes the precomputation of 16 points to speed up scalar multiplication [Knu97, BC89]. If bigger window and more storage for precomputed points are used, the windowing method can be even faster. Pippenger's bucket method and its variants decompose the scalar, then sort out all points into buckets with respect to their scalars, and finally utilize an accumulation algorithm to add them together [Pip76, BDLO12]. Another line of research lies in constructing new number systems to represent the scalar, such as basic digit sets [Mat82, BGMW92] and multi-base number systems [DKS09, SIM12, YWLT13]. Researchers also try to make the addition arithmetic more efficient by using different curve representations, such as projective coordinates and Jacobian coordinates that eliminate the inversion operations, and Montgomery form that only utilizes x-coordinate [Mon87]. Differential addition chains (DACs) are used accompanying with x-only-coordinate systems, for example, PRAC chains [Mon92], DJB chains [Ber06] and other multi-dimensional DACs [Bro15, Rao15]. Most of the aforementioned techniques can be applied to MSM where the number of points is small.

When the number of points in MSM is big, which is the situation in pairing-based trusted setup zkSNARK applications, Pippenger's bucket method and its variants are the state-of-the-art algorithms that outperform other competitors. Bernstein *et al.* [BDLO12] investigated Bos-Coster method [DR94, Section 4], Straus method [Str64] and Pippenger's bucket method, then chose Pippenger's bucket method to implement batching forgery identification for elliptic curve signatures, which marks the beginning of extensive deployment of Pippenger's bucket method for computing MSM with a big number of points.

In practice, all of the popular zkSNARK-oriented implementations, such as Zcash [Zca], TurboPLONK [GJW20], Bellman [Bel], gnark [gna], choose Pippenger's bucket method or its variants to compute MSM over fixed points.

1.2 Our contribution

This paper proposes a new bucket set construction that yields an efficient algorithm to compute MSM over fixed points in the context of Pippenger's bucket method. Our construction targets on n-scalar multiplication with $2^{10} \leq n \leq 2^{21}$, which is desirable for many pairing-based trusted setup zkSNARK applications. Our main contributions are summarized as follows.

- *A new subsum accumulation algorithm.* After sorting out points into buckets with respect to their scalars, Pippenger's bucket method would compute intermediate subsums and utilize an accumulation algorithm to add those subsums together. The original subsum accumulation Algorithm 1 presented in Section 2.3 is applicable for the situation where the scalars in the bucket set are consecutive. When the scalars in the bucket set are inconsecutive, Algorithm 1 would be less efficient. This paper proposes a new subsum accumulation Algorithm 3 that accumulates m intermediate subsums using at most $2m + d - 3$ additions, where d is the maximum difference between two neighbor elements in the bucket set.

- *A construction of bucket set that yields efficient algorithm to compute MSM over fixed points.* The proposed bucket set construction carefully selects integer elements from $[0, q/2]$ so that for all t $(0 \leq t \leq q)$, there exist an integer b in the bucket set and an integer $m \in \{1, 2, 3\}$ such that the following assertion holds,

$$t = mb \text{ or } t = q - mb.$$

  When instantiating over the BLS12-381 curve [Bow17], this construction would yield an algorithm that takes advantage of $3nh$ precomputed points to evaluate the n-scalar multiplication over fixed points where all scalars are smaller than a 255-bit prime r, using at most approximately

$$\begin{cases} (nh + 0.21q) \text{ additions, if } q = 2^c \ (10 \leq c \leq 31, c \neq 15, 16, 17), \\ (nh + 0.28q) \text{ additions, if } q = 2^{16}, \end{cases}$$

  where $h = \lceil \log_q r \rceil$. The theoretical analysis shows that for $n = 2^e$ $(10 \leq e \leq 21)$, the proposed algorithm saves more than 21% computational cost compared to Pippenger's bucket method, and that it saves 2.6% to 9.6% computational cost compared to the most popular variant of Pippenger's bucket method, which is described in Section 2.3.2.

- *The feasibility of accelerating the computation of MSM by taking advantage of large precomputation tables and the effectiveness of our new construction are demonstrated by our implementation.* We implemented the popular variant of Pippenger's bucket method and our construction based on the BLS12-381 library `blst` [bls]. When computing n-scalar multiplication over fixed points in the BLS12-381 curve groups, the experimental result shows that the proposed construction saves more than 17.7% of the computing cost compared to the Pippenger's bucket method implementation built in `blst` for $n = 2^e$ $(10 \leq e \leq 21)$, and that it saves 3.1% to 9.2% of the computing cost compared to the variant of Pippenger's bucket method for $n = 2^e$ $(10 \leq e \leq 21,\ e \neq 16, 20)$.

The paper is organized as follows. In Section 2 several popular MSM algorithms including Pippenger's bucket method and one of its popular variants are reviewed. Then we propose a new subsum accumulation algorithm in Section 3. In Section 4, we present a framework of computing MSM over fixed points taking the advantage of precomputation. This framework is used to derive our new MSM algorithm. Section 5 is dedicated to the construction of our new bucket set and multiplier set. We instantiate our construction over BLS12-381 curve in Section 6 and do the theoretical time complexity analysis. In the end, we present the implementation and experimental result in Section 7.

Let us first introduce the notations used throughout the paper before diving into the content.

Notations. Without special explanations hereinafter, let E be an elliptic curve group and r be its order. Let $\lfloor x \rfloor$ be the largest integer that is equal to or smaller than x, and $\lceil x \rceil$ be the smallest integer that is equal to or greater than x. Let $||$ be bit string concatenation. Notation $S_{n,r}$ represents the following MSM over fixed points,

$$S_{n,r} = a_1 P_1 + a_2 P_2 + ... + a_n P_n, \tag{1}$$

where a_i's are scalars such that $0 \leq a_i < r$ and P_i's are fixed points in E. Radix $q = 2^c$ is an integer used to express a scalar in its radix q representation. Integer h is the length of a scalar in its radix q representation, i.e., $h = \lceil \log_q r \rceil$. The term *addition* refers to the point addition arithmetic in E. Let us assume for simplicity the computational cost of doubling and that of addition in E are the same, denoted as A. This is the norm in Pippenger-like algorithms, where the major operations are additions. The storage size of a point is denoted as P.

2 Recap of multi-scalar multiplication methods

In this section we review several widely used methods that compute $S_{n,r}$ with large n, namely trivial method, Straus method, Pippenger's bucket method and one of the variants of Pippenger's bucket method.

2.1 Trivial method

In trivial method, each a_iP_i in (1) is computed separately by the doubling and addition method, then n intermediate results are added together to obtain the final result. In the worst case each scalar multiplication cost about $2 \cdot (\lceil \log_2 r \rceil - 1) \cdot A$, the total cost of computing $S_{n,r}$ is

$$[2 \cdot (\lceil \log_2 r \rceil - 1) \cdot n + (n-1)] \cdot A \approx 2n \log_2 r \cdot A. \tag{2}$$

If non-adjacent form is used to represent the scalar a_i $(i = 1, 2, ..., n)$, because every non-zero digit has to be adjacent to two 0s, in the worst case there are half non-zero digits in a_i. The cost of each scalar multiplication would drop to about $(3/2)\lceil \log_2(r) \rceil \cdot A$. The time complexity of computing $S_{n,r}$ in the worst case is about

$$\left[\frac{3}{2}\lceil \log_2 r \rceil \cdot n + (n-1)\right] \cdot A \approx \frac{3}{2} \cdot n \log_2 r \cdot A. \tag{3}$$

2.2 Straus method

In order to compute $S_{n,r}$, Straus method [Str64] precomputes 2^{nc} points

$$\{b_1P_1 + b_2P_2 + ... + b_nP_n \mid \text{ for } 0 \le b_i \le 2^c - 1,\ i = 1, 2, ..., n\},$$

where c is a small integer. It then divides each a_i (in its binary form with high order bit to the left) into segments of length c, i.e.,

$$a_i = a_{i,h-1}||a_{i,h-2}||...||a_{i1}||a_{i0} = \sum_{j=0}^{h-1} a_{ij}2^{jc},\ i = 1, 2, ..., n, \tag{4}$$

where $h = \lceil \log_2(r)/c \rceil$, and $0 \le a_{ij} < 2^c$ for $1 \le j \le h-1$. It retrieves the point

$$S_{n,2^c} = a_{1,h-1}P_1 + a_{2,h-1}P_2 + ... + a_{n,h-1}P_n \tag{5}$$

from the precomputation table, doubles it c times, adds the precomputed point

$$a_{1,h-2}P_1 + a_{2,h-2}P_2 + ... + a_{n,h-2}P_n \tag{6}$$

to obtain

$$S_{n,2^{2c}} = (a_{1,h-1}||a_{1,h-2})P_1 + (a_{2,h-1}||a_{2,h-2})P_2 + ... + (a_{n,h-1}||a_{n,h-2})P_n. \tag{7}$$

By repeating such process for $h-1$ times, we obtain

$$\begin{aligned} S_{n,2^{hc}} =& (a_{1,h-1}||a_{1,h-2}||...||a_{10})P_1 + (a_{2,h-1}||a_{2,h-2}||...||a_{20})P_2 \\ &+ ... + (a_{n,h-1}||a_{n,h-2}||...||a_{n,0})P_n. \end{aligned}$$

$S_{n,2^{hc}}$ is exactly what we aim to compute, i.e., $S_{n,r}$.

Straus method is only suitable for small n because when n goes big the precomputation would be exponentially large. One variant that can be used for large number n is to only store $n \cdot (2^c - 1)$ precomputed values,

$$\{b_i P_i \mid 1 \leq b_i \leq 2^c - 1, i = 1, 2, ..., n\},$$

where c is a small integer. At j-th iteration of (5)(6)(7) ($j = 0, 1, 2, ..., h - 1$) in Straus method, separately add together precomputed points $a_{1j}P_1$, $a_{2j}P_2$,..., $a_{nj}P_n$ with $n - 1$ additions to obtain

$$a_{1j}P_1 + a_{2j}P_2 + ... + a_{nj}P_n.$$

The storage size would drop from $2^{nc} \cdot P$ to

$$n(2^c - 1) \cdot P.$$

This process would repeat h times, each time it conducts n additions and c doublings (the last time does not require doubling), so the computational cost is approximately

$$(n + c)h \cdot A. \tag{8}$$

2.3 Pippenger's bucket method

Here we introduce Pippenger's bucket method presented in [BDLO12, Section 4], which is an application of Pippenger's algorithm [Pip76].

Pippenger's bucket method proceeds the same as what Straus method does except for computing

$$S_{n,2^c} = a_{1j}P_1 + a_{2j}P_2 + ... + a_{nj}P_n, \tag{9}$$

where $j = 0, 1, 2, ..., h - 1,\ h = \lceil \log_2(r)/c \rceil$.

Pippenger's bucket method evaluates (9) by first sorting all the points into $(2^c - 1)$ buckets with respect to their scalars. We denote the intermediate subsum of those points corresponding to scalar i as S_i. It computes all $S_i{'}s$ ($i = 1, 2, ..., 2^c - 1$) using at most $(n - (2^c - 1))$ additions. Finally it computes $S_{n,2^c} = \sum_{i=1}^{2^c-1} i \cdot S_i$ by Algorithm 1 using at most $2(2^c - 2)$ additions.

Algorithm 1 Subsum accumulation algorithm I

```
Input: $S_1, S_2, ..., S_m$.
Output: $1S_1 + 2S_2 + ... + mS_m$.
  tmp = 0
  tmp1 = 0
  for i = m to 1 do
      tmp = tmp + $S_i$
      tmp1 = tmp1 + tmp
  return tmp1
```

The correctness of Algorithm 1 is ensured by the following equation,

$$\sum_{i=1}^{m} iS_i = \sum_{i=1}^{m}\sum_{j=1}^{i} S_i = \sum_{j=1}^{m}\sum_{i=j}^{m} S_i.$$

The computation of $S_{n,2^c}$ costs $n - (2^c - 1) + 2(2^c - 2) \approx (n + 2^c)$ additions. The computational cost of $S_{n,r}$ is thus approximately

$$(n + 2^c)h \cdot A. \tag{10}$$

Compared to (8), in the first glimpse it seems that Pippenger's bucket method is less efficient against Straus method, but this might not be right for large n. Because there is no precomputation requirement in Pippenger's bucket method, bigger c can be selected to minimize the overall computational cost.

2.3.1 The variant

In the aforementioned Pippenger's bucket method, one downside is that Algorithm 1 runs h times. If there is storage available for precomputation, this shortcoming can be circumvented by the variant presented in [BGMW95].

Choose a radix $q = 2^c$, partition a_i $(i = 1, 2, ..., n)$ into segments as follows,

$$a_i = a_{i,h-1}||a_{i,h-2}||...||a_{i0} = \sum_{j=0}^{h-1} a_{ij}q^j, \tag{11}$$

where $h = \lceil \log_q r \rceil$, $0 \le a_{ij} < q$ $(0 \le j \le h-1)$. It follows that

$$\begin{aligned} S_{n,r} &= a_1P_1 + a_2P_2 + ... + a_nP_n \\ &= \sum_{i=1}^{n}(\sum_{j=0}^{h-1} a_{ij}q^j)P_i \\ &= \sum_{i=1}^{n}\sum_{j=0}^{h-1} a_{ij} \cdot q^jP_i \\ &=: S_{nh,q}. \end{aligned} \tag{12}$$

We precompute the following points

$$\{q^jP_i \mid i = 1, 2, ..., n,\ j = 0, 1, 2, ..., h-1\},$$

which requires the storage size of

$$nh \cdot P,$$

then $S_{n,r} = S_{nh,q}$ can be computed by using Algorithm 1 only once. The computational cost is

$$[nh - (q-1) + 2(q-2)] \cdot A \approx (nh + q) \cdot A.$$

2.3.2 Further optimization

Pippenger's bucket method and the variant can be further optimized by halving the size of the bucket set. Let radix $q = 2^c$, using the observation that in an elliptic curve group $-P$ is obtained from P by taking the negative of its y coordinate with almost no cost, all the buckets can be restricted to scalars that are no more than $q/2$ if

$$q^{h-1} < r \le q/2 \cdot q^{h-1},$$

where $h = \lceil \log_q r \rceil$. Algorithm 2 can be used to convert scalar a $(0 \le a < r)$ from its standard q-ary form to the representation where every digit is in the range of $[-q/2, q/2]$.

Algorithm 2 Scalar conversion I

Input: $\{a_j\}_{0\leq j\leq h-1},\ 0 \leq a_j < q$ such that $a = \sum_{j=0}^{h-1} a_j q^j$.

Output: $\{b_j\}_{0\leq j\leq h-1}, -q/2 \leq b_j \leq q/2$ such that $a = \sum_{j=0}^{h-1} b_j q^j$.

```
1: for $j = 0$ to $h-2$ by $1$ do
2:     if $a_j \leq q/2$ then
3:         $b_j = a_j$
4:     else
5:         $b_j = a_j - q$
6:         $a_{j+1} = a_{j+1} + 1$
7: $b_{h-1} = a_{h-1}$
8: return $\{b_j\}_{1\leq j\leq h-1}$
```

The correctness of Algorithm 2 is straightforward. Notice that the assumption ensures $a_{h-1} \leq q/2 - 1$, so $b_{h-1} \leq q/2$ considering the possible carry bit from a_{h-2}.

The time complexity of Pippenger's bucket method would thus drop to

$$h\,(n + q/2) \cdot A, \tag{13}$$

and the complexity of the variant would be

$$(nh + q/2) \cdot A. \tag{14}$$

Henceforward when mentioning Pippenger's bucket method and Pippenger's variant, we refer to the algorithms whose time complexities are (13) and (14) respectively.

2.4 Comparison of multi-scalar multiplication algorithms

We summarize in Table 1 the precomputation storage and the time complexity of computing $S_{n,r}$ by the aforementioned methods together with our construction proposed in Section 5. Here $q = 2^c$, $h = \lceil \log_q r \rceil$. Radix q is selected to minimize the computational cost. The time complexity of Pippenger's bucket set and Pippenger's variant hold if $r \leq q/2 \cdot q^{h-1}$. The time complexity of our construction holds when r/q^h is small.

Table 1: Comparison of different methods that computes $S_{n,r}$

Method	Storage	Worst case complexity
Trivial method	$n \cdot P$	$3/2 \cdot (n \log_2 r) \cdot A$
Straus method [Str64]	$n2^c \cdot P$	$h(n + c) \cdot A$
Pippenger [Pip76, BDLO12]	$n \cdot P$	$h(n + q/2) \cdot A$
Pippenger variant [BGMW95]	$nh \cdot P$	$(nh + q/2) \cdot A$
Our construction	$3nh \cdot P$	$(nh + 0.21q) \cdot A$

3 A new subsum accumulation algorithm

During the computation of $S_{n,r}$ by Pippenger's bucket method, after sorting every point into the bucket with respect to its scalar and computing the intermediate subsum $S_i's$, the reminder is invoking a subsum accumulation algorithm to compute

$$S = b_1 S_1 + b_2 S_2 + \ldots + b_m S_m,$$

where $1 \leq b_1 \leq b_2 \leq \ldots \leq b_m$. When set $\{b_i\}_{1\leq i\leq m}$ is not a sequence of consecutive integers, Algorithm 1 shows the limitation of handling such case with less efficiency. One may utilize Bos-Coster method [DR94, Section 4] to deal with this case but it is a recursive algorithm and its complexity is not easy to analyze. Here we propose a straightforward algorithm to tackle this case.

Define $b_0 = 0$, let

$$d = \max_{1\le i\le m} \{b_i - b_{i-1}\},$$

then S can be computed by Algorithm 3.

Algorithm 3 Subsum accumulation algorithm II

Input: $b_1, b_2, ..., b_m,\ S_1, S_2, ..., S_m$.

Output: $S = b_1S_1 + b_2S_2 + ... + b_mS_m$.

```
1: Define a length-$(d+1)$ array $\texttt{tmp} = [0] \times (\texttt{d}+1)$
2: for $\texttt{i} = \texttt{m}$ to $1$ by $-1$ do
3:     $\texttt{tmp}[0] = \texttt{tmp}[0] + \texttt{S}_\texttt{i}$
4:     $\texttt{k} = \texttt{b}_\texttt{i} - \texttt{b}_{\texttt{i}-1}$
5:     if $\texttt{k} >= 1$ then
6:         $\texttt{tmp}[\texttt{k}] = \texttt{tmp}[\texttt{k}] + \texttt{tmp}[0]$
7: return $1 \cdot \texttt{tmp}[1] + 2 \cdot \texttt{tmp}[2] + ... + \texttt{d} \cdot \texttt{tmp}[\texttt{d}]$
```

Denote $\delta_j = b_j - b_{j-1}$, then $b_i = \sum_{j=1}^{i} \delta_j$. The correctness of Algorithm 3 comes from the following equation,

$$\begin{aligned}\sum_{i=1}^{m} b_iS_i &= \sum_{i=1}^{m}(\sum_{j=1}^{i} \delta_j)S_i \\ &= \sum_{j=1}^{m} \delta_j(\sum_{i=j}^{m} S_i) \\ &= \sum_{k=1}^{d} k \sum_{j=1,\delta_j=k}^{m} (\sum_{i=j}^{m} S_i).\end{aligned} \tag{15}$$

During the execution of Algorithm 3, temp variable $\texttt{tmp}[0]$ stores $\sum_{i=j}^{m} S_i$ when loop index i equals j, and temp variable $\texttt{tmp}[\texttt{k}]$ stores $\sum_{j=1,\delta_j=k}^{m}(\sum_{i=j}^{m} S_i)$ for $1 \le k \le d$ after the **for** loop.

If $\{b_i\}_{1\le i\le m}$ is strictly increasing and $\texttt{k}$ in line 4 goes through $\{1, 2, ..., d\}$, then in the **for** loop (line 2 – 6), each iteration executes exactly 2 additions. Since all $d+1$ temp variables in $\texttt{tmp}$ are initialized as 0's, there are $d+1$ additions with addend 0, which have no computational cost, so the **for** loop executes $2m - (d+1)$ additions. Line 7 is computed by subsum accumulation Algorithm 1 with $2(d-1)$ additions. In total, the cost of Algorithm 3 is $2m + d - 3$ additions.

If $\{b_i\}_{1\le i\le m}$ are not strictly increasing, which means sometimes $\texttt{k}$ in line 4 equals 0, in the corresponding **for** iteration it will only execute one addition by skipping **if** part.

If $\texttt{k}$ in line 4 does not go through all integers in $\{1, 2, ..., d\}$, there exists a $\texttt{tmp}[\texttt{k}]$ $(1 \le \texttt{k} \le d)$ who would skip the **for** loop and stay at 0. In the **for** loop, the addition saved by the fact that $\texttt{tmp}[\texttt{k}]$ is initialized as 0 will no longer be saved. In the mean time when line 7 is executed, at least one addition will be saved because $\texttt{tmp}[\texttt{k}] = 0$, so the total cost will not increase.

To sum up, the cost of Algorithm 3 in the worst case is $(2m + d - 3) \cdot A$. When $d = 1$, Algorithm 3 degenerates to Algorithm 1.

4 A framework of computing multi-scalar multiplication over fixed points

The following framework is inspired by Brickell *et al.* [BGMW95], who presented a similar method to compute single scalar multiplication using the notion of basic digit sets. They did not consider the possible overflow of the most significant digit of a scalar, which is not a big issue in single scalar multiplication while it matters in MSM $S_{n,r}$, because the overflow will increase the computational cost by at most n additions. Here we give a straightforward illustration to the framework without the involvement of basic digit sets.

Suppose we are going to compute $S_{n,r}$. Let M be a set of integers, B be a set of non-negative integers and $0 \in B$. Given scalar a_i $(0 \leq a_i < r)$ in its radix q representation

$$a_i = \sum_{j=0}^{h-1} a_{ij} q^j,$$

where $h = \lceil \log_q r \rceil$, if every a_{ij} $(1 \leq i \leq n,\ 0 \leq j \leq h-1)$ is the product of an element from set M and an element from set B, i.e.,

$$a_{ij} = m_{ij} b_{ij}, m_{ij} \in M, b_{ij} \in B,$$

$S_{n,r}$ can be computed as follows,

$$\begin{aligned} S_{n,r} &= \sum_{i=0}^{n} a_i P_i = \sum_{i=1}^{n} (\sum_{j=0}^{h-1} a_{ij} q^j) P_i \\ &= \sum_{i=1}^{n} (\sum_{j=0}^{h-1} m_{ij} b_{ij} q^j) P_i = \sum_{i=1}^{n} \sum_{j=0}^{h-1} b_{ij} \cdot m_{ij} q^j P_i. \end{aligned} \tag{16}$$

Denote $P_{ij} = m_{ij} q^j P_i$, then

$$\begin{aligned} S_{n,r} &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} b_{ij} P_{ij} \\ &= \sum_{i=1}^{n} \sum_{j=0}^{h-1} (\sum_{k \in B} k \cdot \sum_{i,j \text{ s.t. } b_{ij}=k} P_{ij}) \\ &= \sum_{k \in B} k \cdot (\sum_{i=1}^{n} \sum_{j=0}^{h-1} \sum_{i,j \text{ s.t. } b_{ij}=k} P_{ij}). \end{aligned} \tag{17}$$

Suppose those $nh|M|$ points

$$\{m q^j P_i \mid 1 \leq i \leq n, 0 \leq j \leq h-1, m \in M\} \tag{18}$$

are precomputed, and define intermediate subsum S_k,

$$S_k = \sum_{i=1}^{n} \sum_{j=0}^{h-1} \sum_{i,j \text{ s.t. } b_{ij}=k} P_{ij}, \quad k \in B.$$

Equation (17) can be evaluated by first computing all $S_k's$ $(k \in B)$ with at most $nh - (|B|-1)$ additions, the reason is straightforward since there are nh points being sorted into $|B|-1$ subsums. The reminder is computed by Algorithm 3 with at most $2(|B|-1)+d-3$ additions, where d is the maximum difference between two neighbor elements in B.

To sum up, the worst case time complexity of computing $S_{n,r}$ is

$$(nh + |B| + d - 4) \cdot A, \tag{19}$$

where $h = \lceil \log_q r \rceil$, with the help of

$$nh|M| \tag{20}$$

precomputed points.

Set M is called a *multiplier set*, because the set of precomputed points contains the points multiplied by every element from M. Set B is called a *bucket set*, since all points are sorted into subsum buckets with respect to the scalars in B. This framework is translated into Algorithm 4.

Algorithm 4 Multi-scalar multiplication over fixed points

```
Input: Scalars $a_1, a_2, ..., a_n$, fixed points $P_1, P_2, ..., P_n$, radix $q$, scalar length $h$, multiplier set $M = \{m_0, m_1, ..., m_{|M|-1}\}$, bucket set $B = \{b_0, b_1, ..., b_{|B|-1}\}$.
Output: $S_{n,r} = \sum_{i=1}^{n} a_i P_i$.
1: Precompute a length-$nh|M|$ point array precomputation, such that
       $\texttt{precomputation}[|\texttt{M}|((\texttt{i}-1)\texttt{h}+\texttt{j})+\texttt{k}] = \texttt{m}_\texttt{k}\texttt{q}^\texttt{j}\texttt{P}_\texttt{i}$.
   Precompute a hash table mindex to record the index of every multiplier, such that $\texttt{mindex}[\texttt{m}_\texttt{k}] = \texttt{k}$. Precompute a hash table bindex to record the index of every bucket, such that $\texttt{bindex}[\texttt{b}_\texttt{k}] = \texttt{k}$.
2: Convert every $a_i$ to its standard $q$-ary form, then convert it to $\texttt{a}_\texttt{i} = \sum_{\texttt{j}=0}^{\texttt{h}-1} \texttt{m}_{\texttt{ij}}\texttt{b}_{\texttt{ij}}\texttt{q}^\texttt{j}$.
3: Create a length-$nh$ scalar array scalars, such that $\texttt{scalars}[(\texttt{i}-1)\texttt{h}+\texttt{j}] = \texttt{b}_{\texttt{ij}}$. Create a length-$nh$ array points recording the index of points, such that $\texttt{points}[(\texttt{i}-1)\texttt{h}+\texttt{j}] = |\texttt{M}|((\texttt{i}-1)\texttt{h}+\texttt{j}) + \texttt{mindex}[\texttt{m}_{\texttt{ij}}]$. $n$-scalar multiplication $S_{n,r}$ is equivalent to the following $nh$-scalar multiplication
       $\sum_{\texttt{i}=0}^{\texttt{nh}-1} \texttt{scalars}[\texttt{i}] \cdot \texttt{precomputation}[\texttt{points}[\texttt{i}]]$,
   where every scalar in scalars is from bucket set $B$.
4: Create a length-$|B|$ point array buckets to record the intermediate subsums, and initialize every point to infinity. For $0 \le \texttt{i} \le \texttt{nh}-1$, add point $\texttt{precomputation}[\texttt{points}[\texttt{i}]]$ to bucket $\texttt{buckets}[\texttt{bindex}[\texttt{scalars}[\texttt{i}]]]$.
5: Invoke Algorithm 3 to compute $\sum_{\texttt{i}=0}^{|\texttt{B}|-1} \texttt{b}_\texttt{i} \cdot \texttt{buckets}[\texttt{i}]$, return the result.
```

If we denote the expected number of zero element in the length-nh array `scalars` as f, and assume all elements in the length-$|B|$ array `buckets`, Step 5 are none-zero, then the average cost can be estimated as

$$(nh + |B| + d - f) \cdot A. \tag{21}$$

From (19)(21) we can see, given n and r, in order to reduce the time complexity of computing $S_{n,r}$, we can choose a larger radix q to make h smaller, or find a smaller bucket set B. Those two alternatives are closely related.

Here are two examples of utilizing the framework.

Example 1. Under this framework, Pippenger's variant presented in Section 2.3.2 has

$$M = \{-1, 1\}, \ B = \{0, 1, 2, ..., 2^{c-1}\}.$$

Example 2. For radix $q = 2^c$ such that

$$q^{h-1} < r \le 1/4 \cdot q \cdot q^{h-1},$$

we denote $\lambda = q \bmod 3$, $\lambda \in \{1, 2\}$. The multiplier set is picked as

$$M = \{1, -1, 3, -3\}, \tag{22}$$

the corresponding bucket set is

$$B = \{i \mid 0 \leq i \leq \frac{q}{4}\} \cup \{3i - \lambda \mid \text{for all } i \text{ s.t. } \frac{q}{4} \leq 3i - \lambda \leq \frac{q}{2}\}. \tag{23}$$

It can be shown that this is a valid construction and that $|B| \leq q/3 + 2$, thus the pair (M, B) yields an algorithm to compute $S_{n,r}$ using at most

$$(nh + \left\lceil \frac{q}{3} \right\rceil) \cdot A.$$

5 A construction of multiplier set and bucket set

In this section, we construct a pair of multiplier set and bucket set (M, B) that can be utilized to speed up the computation of $S_{n,r}$ under the framework presented in Section 4. The essential difficulty to reduce the size of B is to make sure that every scalar in $[0, r-1]$ can be converted to its radix q representation where every digit is the product of an element from M and an element from B.

Given a scalar a $(0 \leq a < r)$ in its standard q-ary representation

$$a = \sum_{j=0}^{h-1} a_j q^j,\ 0 \leq a_j < q,\ 0 \leq j \leq h-1, \tag{24}$$

we will show that our construction enables the scalar conversion from its standard q-ary form to the required radix q representation, thus yielding efficient $S_{n,r}$ computation algorithm.

5.1 Our construction

For radix $q = 2^c$ $(10 \leq c \leq 31)$, the multiplier set is picked as

$$M = \{1, 2, 3, -1, -2, -3\}. \tag{25}$$

Bucket set B is established by an algorithm.

In order to determine B, let us first define three auxiliary sets B_0, B_1 and B_2. Let $r_{h-1} = \lfloor r/q^{h-1} \rfloor$ be the maximum leading term of a scalar in its standard q-ary expression,

$$\begin{aligned} B_0 =& \{0\} \cup \{b \mid 1 \leq b \leq q/2, s.t.\ \omega_2(b) + \omega_3(b) \equiv 0 \bmod 2\}, \\ B_2 =& \{0\} \cup \{b \mid 1 \leq b \leq r_{h-1} + 1, s.t.\ \omega_2(b) + \omega_3(b) \equiv 0 \bmod 2\}, \end{aligned} \tag{26}$$

where $\omega_2(b)$ represents the exponent of factor 2 in b, and $\omega_3(b)$ represents the exponent of factor 3 in b. For instance, if $i = 2^e k,\ 2 \nmid k$, $\omega_2(b) = e$. From the definitions, B_0 (or B_2) has such a property that for all $0 \leq t \leq q/2$ (or $0 \leq t \leq r_{h-1} + 1$), there exist an element $b \in B_0$ (or $b \in B_2$) and an integer $m \in \{1, 2, 3\}$, such that

$$t = mb.$$

Set B_0 itself is a valid bucket set construction, which was also mentioned in [BGMW95]. Since we can utilize negative elements in M, there are redundant elements to be removed from B_0. Set B_1 is defined by Algorithm 5, and the following Property 1 holds for B_1.

Algorithm 5 Construction of auxiliary set B_1

Input: $B_0,\ q$.
Output: B_1.

```
1: $B_1 = B_0$
2: for $i = q/4$ to $q/2 - 1$ by 1 do
3:     if $i$ is in $B_0$ and $q - 2 \cdot i$ is in $B_0$ then
4:         $B_1$.remove($q - 2 \cdot i$)
5: for $i = \lfloor q/6 \rfloor$ to $q/4 - 1$ by 1 do
6:     if $i$ is in $B_0$ and $q - 3 \cdot i$ is in $B_0$ then
7:         $B_1$.remove($q - 3 \cdot i$)
8: return $B_1$
```

Property 1. Given $q = 2^c$ $(10 \le c \le 31)$, for all t $(0 \le t \le q)$, there exist an element $b \in B_1$ and an integer $m \in \{1, 2, 3\}$, such that

$$t = mb \text{ or } t = q - mb.$$

This property is verified by computation using Algorithm 7. It is also asserted by computation that exchanging two *for* loops in Algorithm 5 would construct the same B_1.

Finally the bucket set is proposed to be

$$B = B_1 \cup B_2. \tag{27}$$

Property 2. For the multiplier set M and the bucket set B defined in (25) (27), scalar a $(0 \le a < r)$ can be expressed (not necessarily uniquely) as follows

$$a = \sum_{j=0}^{h-1} m_j b_j q^j,\ m_j \in M, b_j \in B,\ 0 \le j \le h-1. \tag{28}$$

Proof. By Property 1 we know that for arbitrary integer $t \in [0, q]$, it can be expressed as

$$t = mb + \alpha q,\ m \in M,\ b \in B, \alpha \in \{0, 1\},$$

and by the definition of B_2 we know that for integer $t \in [0, r_{h-1} + 1]$, it can be expressed as

$$t = mb,\ m \in \{1, 2, 3\},\ b \in B.$$

Back to Property 2, Algorithm 6 can be used to convert a from its standard q-ary representation defined in (24) to its radix q representation defined in (28).

Algorithm 6 Scalar conversion II

Input: $\{a_j\}_{0 \le j \le h-1},\ 0 \le a_j < q$ such that $a = \sum_{j=0}^{h-1} a_j q^j$.
Output: $\{(m_j, b_j)\}_{0 \le j \le h-1}, m_j \in M, b_j \in B$ such that $a = \sum_{j=0}^{h-1} m_j b_j q^j$.

```
1: for $j = 0$ to $h - 2$ by 1 do
2:     Obtain $m_j, b_j, \alpha_j$ such that $a_j = m_j b_j + \alpha_j q$
3:     $a_{j+1} = \alpha_j + a_{j+1}$
4: Obtain $m_{h-1}, b_{h-1}$ such that $a_{h-1} = m_{h-1} b_{h-1}$
5: return $\{(m_j, b_j)\}_{0 \le j \le h-1}$
```

The correctness of Algorithm 6 comes from the fact that
i) $a_0 \in [0, q-1]$,
ii) $\alpha_j + a_{j+1} \in [0, q]$ for all $0 \le j \le h-3$,
iii) $\alpha_{h-2} + a_{h-1} \in [0, r_{h-1} + 1]$.

□

For every $t \in [0, q]$, a hash table H is precomputed to store its decomposition, i.e., $H(t) = (m, b, \alpha)$ such that $t = mb + \alpha q, m \in M, b \in B, \alpha \in \{0, 1\}$. Steps 2 and 4 in Algorithm 6 are executed by retrieving the decomposition from the hash table. For the proposed (M, B), hash table H can be realized by a length-$(q + 1)$ array `decomposition` using Algorithm 7. `decomposition` is also utilized to verify Property 1 by checking whether in `decomposition` there is any entry whose last element is -1.

Algorithm 7 Construction of digit decomposition hash table

Input: M, B defined in (25) (27).

Output: Length-$(q + 1)$ array `decomposition`, a realization of hash table H.

```
1: Define a length-$(q+1)$ array decomposition and initiate every entry to be $[0, 0, -1]$.
2: for $m \in \{-1, -2, -3\}$ do
3:     for $b \in B$ do
4:         if $m \cdot b + q \geq 0$ then
5:             decomposition$[m \cdot b + q] = [m, b, 1]$
6: for $m \in \{1, 2, 3\}$ do
7:     for $b \in B$ do
8:         if $m \cdot b \leq q$ then
9:             decomposition$[m \cdot b] = [m, b, 0]$
10: return decomposition
```

When instantiating over BLS12-381 curve, it is calculated approximately

$$|B| = \begin{cases} 0.21q, & q = 2^c \ (10 \leq c \leq 31, c \neq 15, 16, 17), \\ 0.28q, & q = 2^{16}. \end{cases} \tag{29}$$

See Section 6 and Appendix A for the detailed parameters. It is checked that the maximum difference between two neighbor elements in B is $d = 6$.

For a point $P = (x, y)$ on elliptic curve E with short Weierstrass form, $-P = (x, -y)$ can be obtained for almost no cost, hence the points associated with negative elements in M can be excluded from the precomputation table. Correspondingly, in Step 3, Algorithm 4, a length-nh boolean array is added to record the sign of multipliers. In Step 4, if a multiplier is negative, the corresponding point should be deducted from the intermediate subsum.

By the computational cost estimation formula presented in (19), we have the following Proposition 1.

Proposition 1. *Given number of points n and group order r, suppose $q = 2^c$ $(10 \leq c \leq 31)$ and $h = \lceil \log_q r \rceil$, the multiplier set and bucket set defined in (25) (27) yield an algorithm to compute MSM $S_{n,r}$ over BLS12-381 curve using at most approximately*

$$\begin{cases} (nh + 0.21q) \cdot A, & q = 2^c \ (10 \leq c \leq 31, c \neq 15, 16, 17), \\ (nh + 0.28q) \cdot A, & q = 2^{16}, \end{cases} \tag{30}$$

with the help of $3nh$ precomputed points

$$\left\{ mq^j P_i \mid 1 \leq i \leq n, \ 0 \leq j \leq h - 1, \ m \in \{1, 2, 3\} \right\}.$$

6 Instantiation

In this section we instantiate our construction over BLS12-381 curve [BLS02, Bow17], and present some theoretical analysis against Pippenger's bucket method and Pippenger's variant.

BLS12-381 curve is a pairing-friendly elliptic curve initially designed by Sean Bowe for the cryptocurrency system Zcash [Zca, Bow17]. It is widely deployed in blockchain applications such as Zcash, Ethereum [eth], Chia [chi], DFINITY [dfi], Algorand [alg]. It provides about 126-bit security [Pol78, BD19, GMT20].

BLS12-381 curve is defined by the equation

$$E : y^2 = x^3 + 4$$

over the prime field $\mathbb{F}_p$, where

$$\begin{aligned} p = \texttt{0x1a0111ea397fe69a4b1ba7b6434bacd7} \\ \texttt{64774b84f38512bf6730d2a0f6b0f624} \\ \texttt{1eabfffeb153ffffb9feffffffffaaab} \end{aligned}$$

is the 381-bit field characteristic (in hexadecimal), and its embedding degree is 12. Two subgroups $\mathbb{G}_1 \subset E(\mathbb{F}_p)$ and $\mathbb{G}_2 \subset E(\mathbb{F}_{p^2})$ over which bilinear pairings are defined have the same 255-bit prime order

$$\begin{aligned} r = \texttt{0x73eda753299d7d483339d80809a1d805} \\ \texttt{53bda402fffe5bfeffffffff00000001}. \end{aligned}$$

6.1 Theoretical analysis

Radix q is called optimal if the number of additions required to compute $S_{n,r}$ in the worst case is minimized. The optimal q and its corresponding scalar length h for different methods is summarized in Table 2. The precomputation size presented in this table is in terms of points in $\mathbb{G}_1$ with affine coordinates. The precomputation size over $\mathbb{G}_2$ would double its counterpart over $\mathbb{G}_1$.

Pippenger's bucket method and Pippenger's variant are those two methods introduced in Section 2.3.2. Our construction refers to the proposed construction presented in Section 5.

Table 2: Radix q, length h and precomputation size utilized to compute $S_{n,r}$

	Pippenger			Pippenger variant			Our construction		
n	q	h	Storage	q	h	Storage	q	h	Storage
2^{10}	2^{8}	32	96.0 KB	2^{12}	22	2.06 MB	2^{13}	20	5.62 MB
2^{11}	2^{10}	26	192 KB	2^{13}	20	3.75 MB	2^{14}	19	10.6 MB
2^{12}	2^{10}	26	384 KB	2^{13}	20	7.50 MB	2^{14}	19	21.3 MB
2^{13}	2^{11}	24	768 KB	2^{14}	19	14.2 MB	2^{16}	16	36.0 MB
2^{14}	2^{12}	22	1.50 MB	2^{16}	16	24.0 MB	2^{16}	16	72.0 MB
2^{15}	2^{13}	20	3.00 MB	2^{16}	16	48.0 MB	2^{16}	16	144 MB
2^{16}	2^{13}	20	6.00 MB	2^{16}	16	96.0 MB	2^{19}	14	252 MB
2^{17}	2^{16}	16	12.0 MB	2^{18}	15	180 MB	2^{20}	13	468 MB
2^{18}	2^{16}	16	24.0 MB	2^{19}	14	336 MB	2^{20}	13	936 MB
2^{19}	2^{16}	16	48.0 MB	2^{20}	13	624 MB	2^{20}	13	1.83 GB
2^{20}	2^{16}	16	96.0 MB	2^{20}	13	1.22 GB	2^{22}	12	3.38 GB
2^{21}	2^{19}	14	192 MB	2^{22}	12	2.25 GB	2^{22}	12	6.75 GB

The number of additions taken to compute $S_{n,r}$ in the worst case and their comparison are summarized in Table 3, where

- Improv1 = ((Our construction) - Pippenger)/Pippenger,
- Improv2 = ((Our construction) - (Pippenger variant))/(Pippenger variant).

Table 3 shows that theoretically when computing $S_{n,r}$ over BLS12-381 curve for $n = 2^e$ $(10 \leq e \leq 21)$, our construction saves 21% to 40% additions compared to Pippenger's bucket method, and it saves 2.6% to 9.6% additions compared to Pippenger's variant.

Table 3: Comparison of number of additions taken to compute $S_{n,r}$ in the worst case

n	Pippenger	Pippenger variant	Our construction	Improv1	Improv2
2^{10}	3.69×10^4	2.46×10^4	2.22×10^4	39.8%	9.6%
2^{11}	6.66×10^4	4.51×10^4	4.23×10^4	36.4%	6.1%
2^{12}	1.20×10^5	8.60×10^4	8.12×10^4	32.2%	5.6%
2^{13}	2.21×10^5	1.64×10^5	1.49×10^5	32.4%	8.8%
2^{14}	4.06×10^5	2.95×10^5	2.80×10^5	30.8%	4.9%
2^{15}	7.37×10^5	5.57×10^5	5.43×10^5	26.4%	2.6%
2^{16}	1.39×10^6	1.08×10^6	1.03×10^6	26.3%	5.0%
2^{17}	2.62×10^6	2.10×10^6	1.92×10^6	26.6%	8.2%
2^{18}	4.72×10^6	3.93×10^6	3.63×10^6	23.1%	7.7%
2^{19}	8.91×10^6	7.34×10^6	7.04×10^6	21.1%	4.1%
2^{20}	1.73×10^7	1.42×10^7	1.35×10^7	22.2%	4.9%
2^{21}	3.30×10^7	2.73×10^7	2.60×10^7	21.2%	4.5%

It is noted that the proposed bucket sets listed in Appendix A are sufficient to compute $S_{n,r}$ over BLS12-381 for $n = 2^e$ $(22 \leq e \leq 28)$. Our method still shows 2.8% $\sim$ 5.8% theoretical improvement against Pippenger's variant in those cases, but its drawback is that the precomputation would be too large.

6.2 Time complexity: worst case versus average case

We are going to show in this section that the difference between the worst case time complexity and the average case is tiny, hence the worst case is used in this paper as the representative. The result relies on the group order r, that is why we do the average case analysis after instantiation. It is done by estimating the expected number of zero elements, denoted as f, in the length-nh array `scalars`, Algorithm 4.

Suppose the group order in its standard q-ary form is $r = \sum_{j=0}^{h-1} r_j q^j$. For every uniformly randomly picked scalar a $(0 \leq a < r)$, when a is converted to the standard q-ary form

$$a = \sum_{j=0}^{h-1} a_j q^j,\ 0 \leq a_j < q,$$

for simplicity we assume that

$$\Pr[a_j = 0] \approx \frac{1}{q},\ \Pr[a_j = (q-1)] \approx \frac{1}{q},\ 0 \leq j \leq h-2,$$

and

$$\Pr[a_{h-1} = 0] = \frac{q^{h-1}}{r} = \frac{q^{h-1}}{\sum_{j=0}^{h-1} r_j q^j} \approx \frac{1}{r_{h-1}+1}.$$

Let us first do the analysis for our construction. When converting scalar a by Algorithm 6 from its standard q-ary form to the radix q representation

$$a = \sum_{j=0}^{h-1} m_j b_j q^j,\ m_j \in M, b_j \in B,\ 0 \le j \le h-1, \tag{31}$$

we know $b_j = 0$ $(1 \le j \le h-2)$ if and only if $a_j = 0$ and the carry bit from the previous digit $\alpha_{j-1} = 0$, or $a_j = q-1$ and the carry bit $\alpha_{j-1} = 1$. Assume the probability of carry bit being 0 is λ, which is equal to the probability of $\alpha = 0$ in array `decomposition` decided by Algorithm 7, then

$$\Pr[b_j = 0] = \lambda\frac{1}{q} + (1-\lambda)\frac{1}{q} = \frac{1}{q},\ 1 \le j \le h-2,$$

and

$$\Pr[b_0 = 0] = \frac{1}{q},\ \Pr[b_{h-1} = 0] = \lambda \cdot \frac{1}{r_{h-1}+1}.$$

If a scalar is converted to the representation in (31), the expectation of number of j's such that $b_j = 0$ is

$$\frac{h-1}{q} + \frac{\lambda}{r_{h-1}+1}.$$

When running Algorithm 4, the expectation of number of zeros in `scalars` is

$$f = \frac{n(h-1)}{q} + \frac{\lambda \cdot n}{r_{h-1}+1}. \tag{32}$$

Define

$$I = \frac{\text{worst case time complexity} - \text{average case time complexity}}{\text{worst case time complexity}}$$

to measure the difference between the worst case time complexity and the average case time complexity. Our method utilizes radix q comparable to n, so $n(h-1)/q$ is a small number that can be ignored. It follows that

$$I = \frac{f}{nh+|B|+2} \approx \frac{\lambda \cdot n/(r_{h-1}+1)}{nh+|B|+2} < \frac{\lambda \cdot n/(r_{h-1}+1)}{nh} = \frac{\lambda}{(r_{h-1}+1)h}. \tag{33}$$

For radix $q = 2^c$ $(10 \le c \le 22, c \ne 15, 17)$ used in Table 2, the triad (q, h, r_{h-1}) can be found in Appendix A, and it is checked that $\lambda < 0.7$ for those radixes. It follows that $I < 1\%$, which means the difference is small.

A similar analysis also applies to Pippenger's bucket method and Pippenger's variant, and (33) still holds for them. Those two methods have $\lambda \approx 0.5$. For $q = 2^c$ $(8 \le c \le 22)$, I is even smaller compared to our construction.

7 Implementation

In order to assess the cost of scalar conversion and the impact of memory locality issue caused by large precomputation size, we conducted the experiment on the basis of **`blst`**, a BLS12-381 signature library written in C and assembly [bls]. **`blst`** library includes the addition/doubling arithmetic and the implementation of Pippenger's bucket method over $\mathbb{G}_1$ and $\mathbb{G}_2$. We implemented Pippenger's variant and our construction following Algorithm 4, we invoked the Pippenger's bucket method implementation built in **`blst`**.

7.1 Implementation analysis

In terms of the scalar conversion, which is Step 2 of Algorithm 4, a scalar is given as a length-8 `uint32_t` array. Both Pippenger's bucket method and Pippenger's variant need to first convert the scalar to its standard q-ary form, then convert to the expression where the absolute value of every digit is no more than $q/2$ using Algorithm 2. Our construction first converts the scalar to its standard q-ary form, then converts to the expression where every digit is the product of an element from the multiplier set and an element from the bucket set using Algorithm 6 with the help of the decomposition hash table. We utilize a length-$(q+1)$ array `decomposition` to realize this hash table. So the concern boils down to retrieving data from array `decomposition`.

In terms of Step 4 in Algorithm 4, where all points are sorted into different buckets, the addition is done between a point fetched from array `precomputation` and another point fetched from array `buckets`. We treat the n fixed points as the length-n `precomputation` array for Pippenger's bucket method. We have the following observations.

- Pippenger's bucket method compute h times the equation (9), so in total it fetches data nh times from its length-n array `precomputation`, it fetches data nh times from its length-$(0.5q+1)$ array `buckets`.
- Pippenger's variant fetches data nh times from its length-nh array `precomputation`, it fetches data nh times from its length-$(0.5q+1)$ array `buckets`.
- Our construction fetches data nh times from its length-$3nh$ array `precomputation`, it fetches data nh times from its array `buckets`, whose length is roughly $0.21q$ ($q \neq 2^c$, $c \in \{15, 16, 17\}$).

Pippenger's variant and our construction show some advantages here regarding the number of fetch operations, since their h's are usually smaller than that of Pippenger's bucket method. Their disadvantages are that the fetch operations are executed in larger `precomputation` and `buckets` arrays. Step 4 of Algorithm 4 is a simple loop, so we utilize prefetch to mitigate the impact of memory access of large arrays.

It is noted that in terms of fetching data from `buckets` array, our construction would have some advantage against Pippenger's variant when the same radix q is used because our construction would use smaller `buckets` array in this case. Even if our radix q ($q \neq 2^{16}$) is twice as big, our construction still keeps such advantage.

7.2 Experimental result

Our experiment is done on an Apple 14 inch MacBook Pro with 3.2 GHz M1 Pro chip, and with 16 GB memory. M1 Pro has advantages like large cache size, high memory bandwidth. Most importantly its cache line is 128 bytes, which is sufficient to accommodate a BLS12-381 $\mathbb{G}_1$ point whose size is 96 bytes. Those characteristics are expected to provide us some benefit when fetching data from large arrays.

The experimental result is presented in Tables 4 and 5. Both Pippenger's variant and our construction use optimal radixes presented in Table 2, while the Pippenger's bucket method built in `blst` utilizes slightly different radixes, explicitly,

$$q = \begin{cases} 2^{e-2} & \text{for } n = 2^e \ (10 \le e \le 12), \\ 2^{e-3} & \text{for } n = 2^e \ (13 \le e \le 21). \end{cases}$$

We keep `blst`'s implementation intact, because on one hand our focus is on the comparison between Pippenger's variant and our construction, on the other hand `blst`'s implementation can serve as a performance benchmark. In Table 4, s.c. v. represents the time spent by Pippenger's variant to do the scalar conversion for all n scalars in $S_{n,r}$, while s.c.

c. represents that of our construction. In Table 5, Improv1 is the comparison between Pippenger's bucket method and our construction, and Improv2 is the comparison between Pippenger's variant and our construction.

Both Pippenger's variant and our construction show a huge improvement compared to Pippenger's bucket method, which demonstrates the feasibility of speeding up the computation of $S_{n,r}$ using large precomputation tables.

If we focus on the comparison between Pippenger's variant and our construction, we have the following observations when computing $S_{n,r}$ in $\mathbb{G}_1$ for $n = 2^e$ $(10 \le e \le 21)$, and in $\mathbb{G}_2$ for $n = 2^e$ $(10 \le e \le 20)$[1],

- In terms of the computation time of scalar conversion, in $\mathbb{G}_1$ Pippenger's variant takes $0.9 \sim 1.5\%$ out of its entire $S_{n,r}$ computation time, while our construction takes $1.1 \sim 2.8\%$. Because in $\mathbb{G}_2$ the addition arithmetic takes relatively more time compared to that in $\mathbb{G}_1$, the percentages are smaller. In $\mathbb{G}_2$ Pippenger's variant takes $0.4 \sim 0.6\%$ out of its whole $S_{n,r}$ computation time, while our construction takes $0.4 \sim 1.1\%$.

- Our construction does not perform good for $n = 2^{16},\ 2^{20}$. For $n = 2^{16}$, our optimal radix $q = 2^{19}$, which is 8 times larger than that of Pippenger's variant. For $n = 2^{20}$, our optimal radix $q = 2^{22}$, which is 4 times larger than that of Pippenger's variant. Since the radix value is even larger than n, it would have negative impact on fetching data from array `buckets` as the analysis in the previous section indicates. When we change to smaller radixes, specifically $q = 2^{18}$ for $n = 2^{16}$, and $q = 2^{20}$ for $n = 2^{20}$, our construction outperforms Pippenger's variant again as the results marked by asterisk show, although theoretically those radixes are not optimal.

- Our construction outperforms Pippenger's variant for $n = 2^e$ $(10 \le e \le 21, e \ne 16,\ 20)$. In those cases, our construction demonstrates $3.1\% \sim 9.2\%$ improvement against Pippenger's variant as Table 5 shows.

Table 4: Experimental time taken to compute $S_{n,r}$ by different methods

			$\mathbb{G}_1$			$\mathbb{G}_2$		
n	s.c. v.	s.c. c.	Pip.	Pip. v.	Constr.	Pip.	Pip. v.	Constr.
2^{10}	123 us	134 us	15.2 ms	9.91 ms	9.03 ms	37.3 ms	24.4 ms	22.2 ms
2^{11}	248 us	265 us	27.1 ms	18.3 ms	17.1 ms	66.4 ms	44.9 ms	41.9 ms
2^{12}	497 us	548 us	48.5 ms	34.2 ms	32.2 ms	119 ms	83.3 ms	78.5 ms
2^{13}	920 us	657 us	89.4 ms	64.8 ms	62.0 ms	221 ms	160 ms	155 ms
2^{14}	1.07 ms	1.23 ms	165 ms	122 ms	114 ms	404 ms	300 ms	279 ms
2^{15}	2.14 ms	2.40 ms	303 ms	224 ms	217 ms	734 ms	541 ms	522 ms
2^{16}	4.29 ms	6.47 ms	551 ms	422 ms	430 ms	1.35 s	1.03 s	1.05 s
$*2^{16}$	4.29 ms	7.63 ms	554 ms	424 ms	418 ms	1.34 s	1.03 s	1.01 s
2^{17}	12.1 ms	14.9 ms	1.06 s	864 ms	822 ms	2.54 s	2.05 s	1.99 s
2^{18}	17.8 ms	28.0 ms	1.93 s	1.60 s	1.49 s	4.69 s	3.88 s	3.61 s
2^{19}	31.3 ms	54.6 ms	3.55 s	2.98 s	2.83 s	8.63 s	7.28 s	6.83 s
2^{20}	62.7 ms	149 ms	6.84 s	5.62 s	5.63 s	16.7 s	13.7 s	13.5 s
$*2^{20}$	62.7 ms	109 ms	6.84 s	5.61 s	5.51 s	16.6 s	13.7 s	13.3 s
2^{21}	120 ms	296 ms	13.2 s	11.2 s	10.7 s	–	–	–

[1]We did not do test in $\mathbb{G}_2$ for $n = 2^{21}$ due to the memory size restriction of the test device.

Table 5: Our method versus Pippenger's bucket method and Pippenger's variant

	$\mathbb{G}_1$		$\mathbb{G}_2$	
n	Improv1	Improv2	Improv1	Improv2
2^{10}	40.6%	8.86%	40.6%	9.26%
2^{11}	36.8%	6.54%	37.0%	6.78%
2^{12}	33.7%	5.78%	34.2%	5.74%
2^{13}	30.7%	4.40%	29.7%	3.13%
2^{14}	31.2%	6.54%	31.0%	7.29%
2^{15}	28.4%	3.19%	29.0%	3.61%
2^{16}	21.9%	-1.88%	21.8%	-2.05%
$*2^{16}$	24.6%	1.48%	24.9%	2.10%
2^{17}	22.1%	4.91%	21.6%	3.08%
2^{18}	22.8%	6.75%	23.0%	7.03%
2^{19}	20.3%	5.12%	20.8%	6.06%
2^{20}	17.7%	-0.13%	18.8%	1.45%
$*2^{20}$	19.4%	1.69%	17.9%	2.66%
2^{21}	19.0%	4.31%	–	–

References

[alg] Algorand: the blockchain for futurefi. https://www.algorand.com/.

[BC89] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *Conference on the Theory and Application of Cryptology*, pages 400–407. Springer, 1989.

[BCG+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, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474. IEEE Computer Society, 2014.

[BD19] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, 32(4):1298–1336, 2019.

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 649–680. Springer, 2021.

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

[Bel] bellman: a crate for building zk-snark circuits. https://github.com/zkcrypto/bellman.

[Ber06] Daniel J Bernstein. Differential addition chains. *URL: http://cr. yp. to/ecdh/diffchain-20060219. pdf*, 2006.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from DARK compilers. 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 677–706. Springer, 2020.

[BGMW92] Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 200–207. Springer, 1992.

[BGMW95] Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation: algorithms and lower bounds. *preprint, Mar*, 27, 1995.

[bls] blst: a bls12-381 signature library focused on performance and security written in c and assembly. https://github.com/supranational/blst.

[BLS02] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *International Conference on Security in Communication Networks*, pages 257–267. Springer, 2002.

[Bow17] Sean Bowe. BLS12-381: New zk-snark elliptic curve construction, 2017.

[Bro15] Daniel R Brown. Multi-dimensional montgomery ladders for elliptic curves, February 17 2015. US Patent 8,958,551.

[CGGN17] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 229–243, 2017.

[chi] Chia network: a better blockchain and smart transaction platform. https://www.chia.net/.

[CHM$^+$19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. *IACR Cryptology ePrint Archive*, 2019:1047, 2019.

[DFGK14] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 532–550. Springer, 2014.

[dfi] Dfinity foundation: Internet computer. https://dfinity.org/.

[DKS09] Christophe Doche, David R Kohel, and Francesco Sica. Double-base number system for multi-scalar multiplications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 502–517. Springer, 2009.

[DR94] 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.

[Ebe] Jacob Eberhardt. Zokrates. https://zokrates.github.io/.

[eth] Ethereum: a technology that's home to digital money, global payments, and applications. https://ethereum.org/en/.

[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, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[GJW20] Ariel Gabizon and Zachary J. Williamson. Proposal: The turbo-plonk program syntax for specifying snark programs. 2020.

[GKR$^+$21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535, 2021.

[GLS11] Steven D Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Journal of cryptology*, 24(3):446–469, 2011.

[GLV01] Robert P Gallant, Robert J Lambert, and Scott A Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Annual International Cryptology Conference*, pages 190–200. Springer, 2001.

[GMT20] Aurore Guillevic, Simon Masson, and Emmanuel Thomé. Cocks–pinch curves of embedding degrees five to eight and optimal ate pairing computation. *Designs, Codes and Cryptography*, 88(6):1047–1081, 2020.

[gna] gnark zk-snark library. https://github.com/ConsenSys/gnark.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2006.

[GOS12] Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *Journal of the ACM*, 59(3):11:1–11:35, 2012.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 444–459. Springer, 2006.

[Gro09] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2009.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[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 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[GS12] Jens Groth and Amit Sahai. Efficient noninteractive proof systems for bilinear groups. *SIAM Journal on Computing*, 41(5):1193–1232, 2012.

[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.

[Knu97] Donald E Knuth. *The Art of Programming, vol. 2 (3rd ed.), Seminumerical algorithms.* Addison Wesley Longman, 1997.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In Ronald Cramer, editor, *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, volume 7194 of *Lecture Notes in Computer Science*, pages 169–189. Springer, 2012.

[Mat82] David W Matula. Basic digit sets for radix representation. *Journal of the ACM (JACM)*, 29(4):1131–1143, 1982.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2111–2128. ACM, 2019.

[Mon87] Peter L Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[Mon92] Peter L Montgomery. Evaluating recurrences of form xm+ n= f (xm, xn, xm-n) via lucas chains, 1983. *Available at ftp. cwi. nl:/pub/pmontgom/Lucas. ps. gz*, 1992.

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

[Pol78] John M Pollard. Monte carlo methods for index computation. *Mathematics of computation*, 32(143):918–924, 1978.

[Rao15] Srinivasa Rao Subramanya Rao. A note on schoenmakers algorithm for multi exponentiation. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 4, pages 384–391. IEEE, 2015.

[SIM12] Vorapong Suppakitpaisarn, Hiroshi Imai, and Edahiro Masato. Fastest multi-scalar multiplication based on optimal double-base chains. In *World Congress on Internet Security (WorldCIS-2012)*, pages 93–98. IEEE, 2012.

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

[YWLT13] Wei Yu, Kunpeng Wang, Bao Li, and Song Tian. Joint triple-base number system for multi-scalar multiplication. In *International Conference on Information Security Practice and Experience*, pages 160–173. Springer, 2013.

[Zca] Zcash: Privacy-protecting digital currency. https://z.cash/.

Appendix

A Our bucket set constructions over BLS12-381 curve

Table 6 lists our bucket set constructions for $q = 2^c$ $(10 \leq c \leq 31,\ c \neq 15, 17)$. Radixes 2^{15} and 2^{17} are abandoned because $|B|/q$ is too large.

Table 6: Bucket sets over BLS12-381 curve

q	h	r_{h-1}	$\|B\|$	d	$\|B\|/q$
2^{10}	26	28	218	6	0.213
2^{11}	24	3	427	6	0.208
2^{12}	22	7	857	6	0.209
2^{13}	20	231	1725	6	0.211
2^{14}	19	7	3417	6	0.209
2^{15}	17	29677	17312	4	0.528
2^{16}	16	29677	18343	6	0.280
2^{17}	15	118710	69249	4	0.528
2^{18}	15	7	54618	6	0.208
2^{19}	14	231	109244	6	0.208
2^{20}	13	29677	220931	6	0.211
2^{21}	13	7	436906	6	0.208
2^{22}	12	7419	874437	6	0.208
2^{23}	12	3	1747625	6	0.208
2^{24}	11	29677	3497731	6	0.208
2^{25}	11	28	6990507	6	0.208
2^{26}	10	1899369	14139299	6	0.211
2^{27}	10	3709	27962333	6	0.208
2^{28}	10	7	55924059	6	0.208
2^{29}	9	7597479	112481229	6	0.210
2^{30}	9	29677	223698691	6	0.208
2^{31}	9	115	447392434	6	0.208