Multi-Scalar-Multiplication on inner curves [DRAFT]

Gautam Botrel and Youssef El Housni

ConsenSys, gnark team
firsname.lastname@consensys.net

Abstract. The bottleneck in the `Prove` algorithm of most of elliptic-curve-based SNARKs is the Multi-Scalar-Multiplication (MSM) algorithm. In this note we give an overview of a variant of the Pippenger MSM algorithm together with a set of optimizations we proposed tailored for inner curves that form a 2-chain (e.g. BLS12-377). This work is implemented as part of a submission to the zprize competition in the open division "Accelerating MSM on Mobile" (`https://www.zprize.io/`).

`https://github.com/gbotrel/zprize-mobile-harness`

1 Introduction

Given a set of n elements $G_1, \cdots, G_n$ (bases) in $\mathbb{G}$ a cyclic group whose order $\#\mathbb{G}$ has b-bit and a set of n integers $a_1, \cdots, a_n$ (scalars) between 0 and $\#\mathbb{G}$, the goal is to compute efficiently the group element $[a_1]G_1 + \cdots + [a_n]G_n$. In SNARK applications, we are interested in large instances of variable-base MSMs ($n = 10^7, 10^8, 10^9$) — with random bases and random scalars — over the groups $\mathbb{G}_1$ and $\mathbb{G}_2$.

The naive algorithm uses a double-and-add strategy to compute each $[a_i]G_i$ then adds them all up, costing on average $3/2 \cdot b \cdot n$ group operations ($+$). There are several algorithms that optimize the total number of group operations as a function of n such as Strauss [13], Bos–Coster [11, Sec. 4] and Pippenger [10] algorithms. For large instances of a variable-base MSM, the fastest approach is a variant of Pippenger's algorithm [3, Section 4]. For simplicity, we call it the bucket-list method.

2 Preliminaries: the bucket-list method

The high-level strategy is in three steps:

- Step 1: reduce the b-bit MSM to several c-bit MSMs for some fixed $c \leq b$
- Step 2: solve each c-bit MSM efficiently
- Step 3: combine the c-bit MSMs into the final b-bit MSM

Step 1: reduce the b-bit MSM to several c-bit MSMs

1. Choose a window $c \leq b$
2. Write each scalar $a_1, \cdots, a_n$ in binary form and partition each into c-bit parts

$$a_i = \underbrace{(a_{i,1}, a_{i,2}, \cdots, \underbrace{a_{i,b/c}}_{c\text{-bit}})_2}_{b\text{-bit}}$$

3. Deduce b/c instances of c-bit MSMs from the partitioned scalars

$$\begin{aligned} T_1 &= [a_{1,1}]G_1 + \cdots + [a_{n,1}]G_n \\ &\vdots \\ T_j &= [a_{1,j}]G_1 + \cdots + [a_{n,j}]G_n \\ &\vdots \\ T_{b/c} &= [a_{1,b/c}]G_1 + \cdots + [a_{n,b/c}]G_n \end{aligned}$$

Cost of Step 1 is negligible.

Algorithm 1: Step 3: combine the c-bit MSMs into the final b-bit MSM

Output: $T = [a_1]G_1 + \cdots + [a_n]G_n$

```
T ← T_1;
for i from 2 to b/c do
    T ← [2^c]T;                    // Double c times
    T ← T + T_i;                   // Add
return T;
```

Step 3: combine the c-bit MSMs into the final b-bit MSM

Cost of Step 3: $(b/c - 1)(c + 1) = b - c + b/c - 1$ group operations.

Step 2: solve each c-bit MSM T_j efficiently

1. For each T_j, accumulate the bases G_i inside buckets
   Each element $a_{i,j}$ is in the set $\{0, 1, 2, \cdots 2^c - 1\}$. We initialize $2^c - 1$ empty buckets (with points at infinity) and accumulate the bases G_i from each T_j

inside the bucket corresponding to the scalar $a_{i,j}$.

$$
\begin{array}{cccccc}
 & & G_k & & & \\
 & & + & & & \\
 & & \vdots & & & \\
 & & + & & & \\
G_{2^c-k} & & G_{23} & & & \\
+ & & + & & & \\
G_7 & G_{15} & G_{19} & & G_{2^c-k'} & \\
+ & + & + & \vdots & + & \\
G_4 & G_3 & G_{18} & & G_1 & \\
\hline
\text{buckets:} & \boxed{1} & \boxed{2} & \boxed{3} & \cdots & \boxed{2^c-1} \\
\text{sum:} & S_1 & S_2 & S_3 & \cdots & S_{2^c-1}
\end{array}
$$

Cost: $n - (2^c - 1) = n - 2^c + 1$ group operations.

2. Combine the buckets to compute T_j
   This step is also a c-bit MSM of size $2^c - 1$ but this time the scalars are ordered and known in advance $S_1 + [2]S_2 + \cdots + [2^c - 1]S_{2^c-1}$, thus we can compute this instance efficiently as follows

$$
\begin{array}{cccccccccc}
 & S_{2^c-1} & & & & & & & & \\
+ & S_{2^c-1} & + & S_{2^c-2} & & & & & & \\
 & \vdots & & & & & & & & \\
+ & S_{2^c-1} & + & S_{2^c-2} & + \cdots + & S_3 & + & S_2 & & \\
+ & S_{2^c-1} & + & S_{2^c-2} & + \cdots + & S_3 & + & S_2 & + S_1 &
\end{array}
$$

$$[2^c - 1]S_{2^c-1} + [2^c - 2]S_{2^c-2} + \cdots + [3]S_3 + [2]S_2 + S_1$$

Cost: $2(2^c - 2) + 1 = 2^{c+1} - 3$ group operations.

Cost of Step 2: $n - 2^c + 1 + 2^{c+1} - 3 = n + 2^c - 2$ group operations.

Combining Steps 1, 2 and 3, the expected overall cost of the bucket-list method is

Total cost: $\frac{b}{c}(n + 2^c) + (b - c - b/c - 1) \approx \frac{b}{c}(n + 2^c)$ group operations.

Remark 1 (On choosing c). The theoretical minimum occurs at $c \approx \log n$ and the asymptotic scaling looks like $(b\frac{n}{\log n})$. However, in practice, empirical choices of c yield a better performance because the memory usage scales with 2^c and there are fewer edge cases if c divides b. For example, with $n = 10^7$ and $b = 256$, we observed a peak performance at $c = 16$ instead of $c = \log n \approx 23$.

3 Optimizations

Parallelism Since each c-bit MSM is independent of the rest, we can compute each (Step 2) on a separate core. This makes full use of up to b/c cores but increases memory usage as each core needs $2^c - 1$ buckets (points). If more than b/c cores are available, further parallelism does not help much because m MSM instances of size n/m cost more than 1 MSM instance of size n.

Precomputation When the bases $G_1, \cdots, G_n$ are known in advance, we can use a smooth trade-off between precomputed storage vs. run time. For each base G_i, choose k as big as the storage allows and precompute k points $[2^c - k]G, \cdots, [2^c - 1]G$ and use the bucket-method only for the first $2^c - 1 - k$ buckets instead of $2^c - 1$. The total cost becomes $\approx \frac{b}{c}(n + 2^c - k)$. However, large MSM instances already use most available memory. For example, when $n = 10^8$ our implementation needs 58GB to store enough BLS12-377 curve points to produce a Groth16 proof. Hence, the precomputation approach yield negligible improvement in our case.

Algebraic structure Since the bases $G_1, \cdots, G_n$ are points in $\mathbb{G}_1$ or $\mathbb{G}_2$, we can use the algebraic structure of elliptic curves to further optimize the bucket-list method.

Non-Adjacent-Form (NAF). Given a point $G_i = (x, y) \in \mathbb{G}_1$ (or $\mathbb{G}_2$), the negative $-G_i$ is $(x, -y)$. This observation is well known to speed up the scalar multiplication $[s]G_i$ by encoding the scalar s in a signed binary form $\{-1, 0, 1\}$ (later called 2-NAF — the first usage might go back to 1989 [9]). However, this does not help in the bucket-list method because the cost increases with the number of possible scalars regardless of their encodings. For a c-bit scalar, we always need $2^c - 1$ buckets. That is said, we can use the 2-NAF decomposition differently. Instead of writing the c-bit scalars in the set $\{0, \cdots, 2^c - 1\}$, we write them in the signed set $\{-2^{c-1}, \cdots, 2^{c-1} - 1\}$ (cf. Alg. 2). If a scalar $a_{i,j}$ is strictly positive we add G_i to the bucket $S_{(a_{i,j})_2}$ as usual, and if $a_{i,j}$ is strictly negative we add $-G_i$ to the bucket $S_{|(a_{i,j})_2|}$. This way we reduce the number of buckets by half.

Total cost: $\approx \frac{b}{c}(n + 2^{c-1})$ group operations.

The signed-digit decomposition cost is negligible but it works only if the bitsize of $\#\mathbb{G}_1$ (and $\#\mathbb{G}_2$) is strictly bigger than b. We use the spare bits to avoid the overflow. This observation should be taken into account at the curve design level.

Curve forms and coordinate systems. To minimize the overall cost of storage but also run time, we store the bases G_i in affine coordinates. This way we only need the tuples (x_i, y_i) for storage (although we can batch-compress these

Algorithm 2: Signed-digit decomposition

Input: $(a_0, \cdots, a_{b/c-1}) \in \{0, \cdots, 2^c - 1\}$

Output: $(a'_0, \cdots, a'_{b/c-1}) \in \{-2^{c-1}, \cdots, 2^{c-1} - 1\}$

for *i from* 0 *to* $b/c - 1$ **do**
  if $a_i \geq 2^{c-1}$ **then**
    assert $i \neq b/c - 1$; // No overflow for the final digit
    $a'_i \leftarrow a_i - 2^c$; // Force this digit into $\{-2^{c-1}, \cdots, 2^{c-1} - 1\}$
    $a_{i+1} \leftarrow a_{i+1} + 1$; // Lend 2^c to the next digit
  else
    $a'_i \leftarrow a_i$
return $(a'_0, \cdots, a'_{b/c-1})$;

following [7]) and we can make use of mixed addition with a different coordinate system.

The overall cost of the bucket-list method is $\frac{b}{c}(n + 2^{c-1}) + (b - c - b/c - 1)$ group operations. This can be broken down explicitly to:

- Mixed additions: to accumulate G_i in the c-bit MSM buckets with cost $\frac{b}{c}(n - 2^{c-1} + 1)$
- Additions: to combine the bucket sums with cost $\frac{b}{c}(2^c - 3)$
- Additions and doublings: to combine the c-bit MSMs into the b-bit MSM with cost $b - c + b/c - 1$
  - $b/c - 1$ additions and
  - $b - c$ doublings

For large MSM instances, the dominating cost is in the mixed additions as it scales with n. For this, we use extended Jacobian coordinates $\{X, Y, ZZ, ZZZ\}$ $(x = X/ZZ, y = Y/ZZZ, ZZ^3 = ZZZ^2)$ trading-off memory for run time compared to the usual Jacobian coordinates $\{X, Y, Z\}$ $(x = X/Z^2, y = Y/Z^3)$ (cf. Table 1).

Coordinate systems	Mixed addition	Addition	Doubling
Jacobian	$7\mathbf{m} + 4\mathbf{s}$	$11\mathbf{m} + 5\mathbf{s}$	$2\mathbf{m} + 5\mathbf{s}$
Extended Jacobian	$8\mathbf{m} + 2\mathbf{s}$	$12\mathbf{m} + 2\mathbf{s}$	$6\mathbf{m} + 4\mathbf{s}$

Table 1. Cost of arithmetic in Jacobian and extended Jacobian coordinate systems. **m**=Multiplication and **s**=Squaring in the field.

Remark 2. In [4], the authors suggest to use affine coordinates for batch addition. That is, they only compute the numerators in the affine addition, accumulate the denominators and then batch-invert them using the Montgomery trick [8]. An affine addition costs $3\mathbf{m} + 1\mathbf{i}$ ($\mathbf{i}$ being a field inversion). For a single addition this is not worth it as $1\mathbf{i} > 7\mathbf{m}$ $(= 10\mathbf{m} - 3\mathbf{m})$. If we accumulate L points and

batch-add them with cost $3L\mathbf{m} + L\mathbf{i} = 6L\mathbf{m} + 1\mathbf{i}$ (the Montgomery trick costing $L\mathbf{i} = 3L\mathbf{m} + 1\mathbf{i}$), this might be worth it. Assuming I=C$\mathbf{m}$, there might be an improvement if we accumulate a number of points $L > C/4$. However, we did not observe a significant improvement in our implementation in `gnark-crypto` compared to the extended Jacobian approach. This is mainly because C is large due the optimized finite field arithmetic in `gnark-crypto` [5]. This means L should be large requiring more memory, not to mention that one needs to implement specialized batch-addition functions in affine coordinates.

We work over fields of large prime characteristic ($\neq 2, 3$), so the elliptic curves in question have always a short Weierstrass form $y^2 = x^3 + ax + b$. Over this form, the fastest mixed addition is achieved using extended Jacobian coordinates. However, there are other forms that enable even faster mixed additions (cf. Table 2).

Form	Coordinates system	Equation	Mixed addition cost
short Weierstrass	extended Jacobian	$y^2 = x^3 + ax + b$	$10\mathbf{m}$
Jacobi quartics	$XXYZZ$, doubling-oriented $XXYZZ$, $XXYZZR$, doubling-oriented $XXYZZR$	$y^2 = x^4 + 2ax^2 + 1$	$9\mathbf{m}$
Edwards	projective, inverted	$x^2 + y^2 = c^2(1 + dx^2y^2)$	$9\mathbf{m}$
twisted Edwards	extended $(XYZT)$ $x = X/Z, y = Y/Z, x \cdot y = T/Z$	$ax^2 + y^2 = 1 + dx^2y^2$	$8\mathbf{m}$
twisted Edwards	extended $(XYZT)$ $x = X/Z, y = Y/Z, x \cdot y = T/Z$	$-x^2 + y^2 = 1 + dx^2y^2$ $(a = -1)$	$7\mathbf{m}$

Table 2. Cost of mixed addition in different elliptic curve forms and coordinate systems assuming $1\mathbf{m} = 1\mathbf{s}$. Formulas and references from [2].

It appears that a twisted Edwards form is appealing for the bucket-list method since it has the lowest cost for the mixed addition in extended coordinates. Furthermore, the arithmetic on this form is *complete*, i.e. the addition formulas are defined for all inputs. This improves the run time by eliminating the need of branching in case of adding the neutral element or doubling compared to a Weierstrass form. We show in Lemma 1 that all inner BLS curves admit a twisted Edwards form.

Proposition 1. *Half of BLS curves are of the form $Y^2 = X^3 + 1$, these are the curves with odd seed x.*

Proof. Let $E : Y^2 = X^3 + b$ be a BLS curve over $\mathbb{F}_p$ and g neither a square nor a cube in $\mathbb{F}_p$. One choice of $b \in \{1, g, g^2, g^3, g^4, g^5\}$ gives a curve with the correct order (i.e. $r \mid \#E(\mathbb{F}_p)$) [12, §X.5]. For all BLS curves, $x - 1 \mid \#E(\mathbb{F}_p)$ [1] and

$3 \mid x-1$ (which leads to all involved parameters being integers). If, additionally, $2 \mid x-1$ then $2, 3 \mid \#E(\mathbb{F}_p)$ and the curve has points of order 2 and 3. A 2-torsion point is $(x_0, 0)$ with x_0 a root of $x^3 + b$, hence $b = (-x_0)^3$ is a cube. The two 3-torsion points are $(0, \pm\sqrt{b})$ hence b is a square. This implies that b is a square and a cube in $\mathbb{F}_p$ and therefore $b = 1$ is the only solution in the set $\{g^i\}_{0 \leq i \leq 5}$ for half of all BLS curves: those with odd x.

Lemma 1. *All inner BLS curves admit a twisted Edwards form $ay^2 + x^2 = 1 + dx^2y^2$ with $a = 2\sqrt{3} - 3$ and $d = -2\sqrt{3} - 3$ over $\mathbb{F}_p$. If further $-a$ is a square, the equation becomes $-x^2 + y^2 = 1 + d'x^2y^2$ with $d' = 7 + 4\sqrt{3}$.*

Proof. Proposition 1 shows that all inner BLS curves are of the form $W_{0,1} : y^2 = x^3 + 1$. The following map

$$\begin{aligned} W_{0,1} &\to E_{a,d} \\ (x, y) &\mapsto \left(\frac{x+1}{y}, \frac{x+1-\sqrt{3}}{x+1+\sqrt{3}}\right) \end{aligned}$$

defines the curve $E_{a,d} : ay^2 + x^2 = 1 + dx^2y^2$ with $a = 2\sqrt{3} - 3$ and $d = -2\sqrt{3} - 3$. The inverse map is

$$\begin{aligned} E_{a,d} &\to W_{0,1} \\ (x, y) &\mapsto \left(\frac{(1+y)\sqrt{3}}{1-y} - 1, \frac{(1+y)\sqrt{3}}{(1-y)x}\right) \end{aligned}$$

If $-a$ is a square in $\mathbb{F}_p$, the map $(x, y) \mapsto (x/\sqrt{-a}, y)$ defines from $E_{a,d}$ the curve $E_{-1,d'}$ of equation $-x^2 + y^2 = 1 + d'x^2y^2$ with $d' = (2\sqrt{3}+3)/(-2\sqrt{3}+3) = 7 + 4\sqrt{3}$.

For the arithmetic, we use the formulas in [6] alongside some optimizations. We take the example of BLS12-377 for which $a = -1$:

- To combine the c-bit MSMs into a b-bit MSM we use unified additions [6, Sec. 3.1] ($9\mathbf{m}$) and dedicated doublings [6, Sec. 3.3] ($4\mathbf{m} + 4\mathbf{s}$).
- To combine the bucket sums we use unified additions ($9\mathbf{m}$) to keep track of the running sum and unified re-additions ($8\mathbf{m}$) to keep track of the total sum. We save $1\mathbf{m}$ by caching the multiplication by $2d'$ from the running sum.
- To accumulate the G_i in the c-bit MSM we use unified re-additions with some precomputations. Instead of storing G_i in affine coordinates we store them in a custom coordinate system (X, Y, T) where $y - x = X$, $y + x = Y$ and $2d' \cdot x \cdot y = T$. This saves $1\mathbf{m}$ and $2\mathbf{m}$ at each addition of G_i.

We note that altough the dedicated addition (resp. the dedicated mixed addition) in [6, Sec. 3.2] saves the multiplication by $2d'$, it costs $4\mathbf{m}$ (resp. $2\mathbf{m}$) to check the operands equality: $X_1Z_2 = X_2Z1$ and $Y_1Z_2 = Y_2Z1$ (resp. $X_1 = X_2Z1$ and $Y_1 = Y_2Z1$). This cost offset makes both the dedicated (mixed) addition and

the dedicated doubling slower than the unified (mixed) addition in the MSM case. We also note that the conversion of all the G_i points given on a short Weierstrass curve with affine coordinates to points on a twisted Edwards curve (also with $a = -1$) with the custom coordinates (X, Y, T) is a one-time computation dominated by a single inverse using the Montgomery batch trick. In proof systems, since the G_i are points from the proving key σ_p, this computation can be part of the `Setup` algorithm and do not impact the `Prove` algorithm.

Our implementation in `gnark-crypto` shows that an MSM instance of size 2^{16} on the BLS12-377 curve is 29% faster when the $\mathbb{G}_i$ points are given on a twisted Edwards curve with the custom coordinates compared to the Jacobian-extended-based version.

4 Conclusion

Multi-scalar-multiplication dominates the proving cost in most elliptic-curve-based SNARKs. The BLS12-377 is an optimized pairing-friendly elliptic curve suitable for both proving generic-purpose statements and in particular for composition and recursive statements. Hence, it is critical to aggresively optimize the computation of MSM instances on BLS12-377. We showed that our work yield a very fast implementation both when the points are given on a short Weierstrass curve and even more when the points are given on a twisted Edwards curve. We showed that this is always the case for inner curves such as BLS12-377 and that the conversion cost is a one-time computation that can be performed in the setup phase.

Open-question: For Groth16, the same scalars a_i are used for both $\mathbb{G}_1$ and $\mathbb{G}_2$ MSMs. We ask if it is possible to mutualize a maximum of computations between these two instances? It seems that moving to a type-2 pairing would allow to deduce the $\mathbb{G}_1$ instance from the $\mathbb{G}_2$ one using an efficient homomorphism over the resulting single point. However, $\mathbb{G}_2$ computations would be done on the much slower full extension $\mathbb{F}_{p^k}$. The pairing, needed for proof verification, would also be slower.

References

1. Barreto, P.S.L.M., Lynn, B., Scott, M.: On the selection of pairing-friendly groups. pp. 17–25. LNCS (2004). https://doi.org/10.1007/978-3-540-24654-1_2
2. Bernstein, D., Lange, T.: Explicit-formulas database. `https://www.hyperelliptic.org/EFD/` (2022)
3. Bernstein, D.J., Doumen, J., Lange, T., Oosterwijk, J.J.: Faster batch forgery identification. pp. 454–473. LNCS (2012). https://doi.org/10.1007/978-3-642-34931-7_26
4. Gabizon, A., Williamson, Z.: Proposal: The turbo-plonk program syntax for specifying snark programs. `https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf` (2020)
5. gnark: Faster big-integer modular multiplication for most moduli. `https://hackmd.io/@gnark/modular_multiplication` (2020)

6. Hisil, H., Wong, K.K.H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. pp. 326–343. LNCS (2008). https://doi.org/10.1007/978-3-540-89255-7_20
7. Koshelev, D.: Batch point compression in the context of advanced pairing-based protocols. Cryptology ePrint Archive, Report 2021/1446 (2021), https://eprint.iacr.org/2021/1446
8. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Mathematics of Computation **48**, 243–264 (1987)
9. Morain, F., Olivos, J.: Speeding up the computations on an elliptic curve using addition-subtraction chains. RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications **24**(6), 531–543 (1990), http://eudml.org/doc/92374
10. Pippenger, N.: On the evaluation of powers and related problems (preliminary version). In: 17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976. pp. 258–263. IEEE Computer Society (1976). https://doi.org/10.1109/SFCS.1976.21, https://doi.org/10.1109/SFCS.1976.21
11. de Rooij, P.: Efficient exponentiation using procomputation and vector addition chains. pp. 389–399. LNCS (1995). https://doi.org/10.1007/BFb0053453
12. Silverman, J.H.: The Arithmetic of Elliptic Curves. Graduate texts in mathematics, Springer, Dordrecht (2009). https://doi.org/10.1007/978-0-387-09494-6
13. Strauss, E.G.: Addition chains of vectors (problem 5125). American Mathematical Monthly **70**(114), 806–808 (1964)