Notes on Multi-Scalar Multiplication (MSM) Optimization

Lizheng Wang 3rd March 2026
lizhengwang1124@gmail.com

1 Introduction to MSM

Multi-Scalar Multiplication (MSM) is a fundamental and often performance-critical operation in cryptography, particularly in the construction of Zero-Knowledge Proofs (ZKPs). The problem is to compute the sum of scalar multiplications of elements in a cyclic group $\mathbb{G}$.

Given a set of group elements $G_1, G_2, \ldots, G_n \in \mathbb{G}$ and a corresponding vector of integer scalars $a = (a_1, a_2, \ldots, a_n)$, the goal of an MSM algorithm is to efficiently compute the sum:

$$S = \sum_{i=1}^{n} a_i \cdot G_i$$

In protocols like Groth16, the prover must compute proofs involving MSM where n can be very large (e.g., 4×10^4 in Zcash). Since the prover's computation time is a major bottleneck in ZKP systems, optimizing the MSM algorithm is crucial for scalability and performance.

A naive approach of computing each scalar multiplication $a_i \cdot G_i$ individually and then summing them results in n scalar multiplication and n group operations (point addition). In general, the scalar a may be any number between 1 and the size $|G|$ of the group. The total number of group operations required to compute the scalar multiplication via the standard double-and-add algorithm is about $3/2 \log |G|$. Cryptographic groups have size at least about 2^{256}. So a scalar multiplication can be roughly 400 times slower than a group operation.[1]

2 The Pippenger Algorithm (Bucket Method)

Pippenger's algorithm is a widely used, bucket-based method for accelerating MSM computations. The core idea is to break down a large MSM problem with b-bit scalars into several smaller MSM problems with c-bit scalars, solve these smaller problems efficiently using buckets, and then combine the results.

2.1 Step 1: Splitting Scalars (Windowing)

First, we select a window size c (where $c \leq b$, with b being the bit-length of the scalars). Each b-bit scalar a_i is split into $m = \lceil b/c \rceil$ chunks or "windows," each of width c bits. We can express each a_i as:

$$a_i = \sum_{j=0}^{m-1} a_{i,j} \cdot 2^{jc}$$

where each coefficient $a_{i,j}$ is a c-bit integer, i.e., $0 \leq a_{i,j} < 2^c$.

By substituting this into the original MSM equation and rearranging the order of summation, we get:

$$S = \sum_{i=1}^{n} \left(\sum_{j=0}^{m-1} a_{i,j} \cdot 2^{jc} \right) G_i = \sum_{j=0}^{m-1} 2^{jc} \left(\sum_{i=1}^{n} a_{i,j} G_i \right)$$

[1] https://a16zcrypto.com/posts/article/17-misconceptions-about-snarks/#section–12

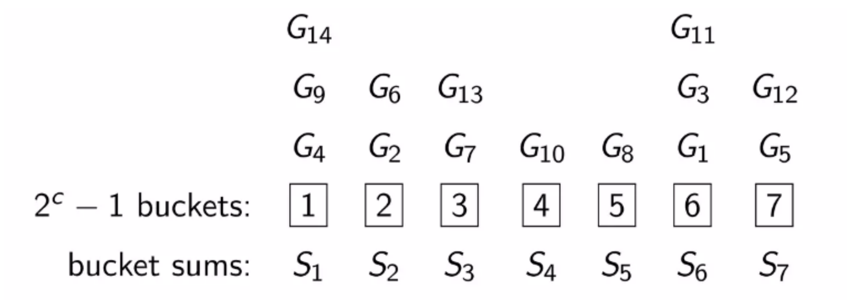

Figure 1: Bucket

Let's define each inner sum as a "sub-MSM" term T_j:

$$T_j = \sum_{i=1}^{n} a_{i,j} G_i$$

Now, the original sum S can be computed by combining these terms:

$$S = T_0 + 2^c T_1 + 2^{2c} T_2 + \cdots + 2^{(m-1)c} T_{m-1}$$

This can be computed efficiently from T_{m-1} down to T_0 using a Horner's method-like approach: start with $S = T_{m-1}$, then repeatedly compute $S \leftarrow (S \cdot 2^c) + T_{j-1}$ for $j = m-1, \ldots, 1$. This combination requires $m-1$ scalar multiplications by 2^c and $m-1$ additions, which result in $\boxed{c(m-1) + m - 1 = b - c + b/c - 1}$ group operation.

2.2 Step 2: Processing c-bit MSMs with Buckets

The next task is to compute each sub-MSM term T_j. Since the scalars $a_{i,j}$ are small ($< 2^c$), we can use the bucket method. For each T_j:

1. **Initialization**: Create $2^c - 1$ empty "buckets," indexed from 1 to $2^c - 1$.
2. **Distribution**: Iterate through all group elements G_i from $i = 1$ to n. For each G_i, look at its corresponding c-bit scalar $a_{i,j}$. If $a_{i,j} = k \neq 0$, add the group element G_i to the k-th bucket.
3. **Aggregation**: After distributing all elements, compute the sum of the points in each bucket. Let S_k be the sum of all points in bucket k.

$$S_k = \sum_{i \text{ s.t. } a_{i,j}=k} G_i$$

   This requires $\boxed{n - (2^c - 1)}$ group additions.

   Once all bucket sums S_k are computed, the sub-MSM term T_j is given by:

$$T_j = \sum_{k=1}^{2^c-1} k \cdot S_k$$

2.3 Step 3: Efficient Summation of Buckets

The final sum for T_j can be computed more efficiently than a naive MSM. The trick is to rewrite the sum as follows:

$$T_j = 1 \cdot S_1 + 2 \cdot S_2 + \cdots + (2^c - 1)S_{2^c-1} = \sum_{k=1}^{2^c-1} \left(\sum_{l=k}^{2^c-1} S_l \right)$$

This can be computed iteratively:

1. Let $B_{2^c-1} = S_{2^c-1}$.
2. For $k = 2^c - 2$ down to 1, compute $B_k = B_{k+1} + S_k$.
3. The final result is $T_j = \sum_{k=1}^{2^c-1} B_k$.

This process requires approximately $2^c - 2$ additions to compute the partial sums B_k and another $2^c - 2$ additions to compute the final sum, for a total of roughly $\boxed{2^{c+1} - 3}$ additions.

2.4 Overall Complexity

Combining the costs, the total number of group additions is approximately $\boxed{\frac{b}{c} \times (n + 2^c)}$. To minimize this expression, the optimal window size c should be chosen such that $n \approx 2^c$, or $c \approx \log_2 n$. With this choice, the complexity of Pippenger's algorithm becomes:

$$O\left(\frac{bn}{\log n}\right)$$

This provides a significant improvement over the naive $O(bn)$ approach.

3 Further Optimizations

3.1 Signed Bucket Indexes (NAF)

The number of buckets can be halved by using a signed digit representation for the c-bit scalars, because of the endomorphism of the elliptic curve group. This is also known as the Non-Adjacent Form (NAF) method. Instead of coefficients in $\{0, \ldots, 2^c - 1\}$, we can recode them into the set $\{-(2^{c-1} - 1), \ldots, -1, 1, \ldots, 2^{c-1} - 1\}$, skipping zero. This reduces the number of buckets from $2^c - 1$ to $2^{c-1} - 1$.

The bucket placement rule is modified:

- If a scalar coefficient $a_{i,j}$ is positive, place G_i into bucket $a_{i,j}$.
- If a scalar coefficient $a_{i,j}$ is negative, place the inverse of the group element, $-G_i$, into bucket $|a_{i,j}|$.

Since finding the inverse of an elliptic curve point (x, y) is trivial (it's just $(x, -y)$), this optimization comes at virtually no cost while halving the work in the bucket summation stage.

3.2 Batch Addition and Inversion

A major cost in elliptic curve cryptography is field inversion, which is required for each point addition when using affine coordinates. When calculating the bucket sums (e.g., $S_k = P_1 + P_2 + P_3 + \ldots$), many independent additions are performed.

Montgomery's batch inversion trick can be used to perform n inversions for the cost of approximately one inversion and $3n$ multiplications. The idea is to compute the inverse of the product of all numbers to be inverted, and then multiply back to find individual inverses. This significantly speeds up the aggregation step within buckets.

3.3 GLV Decomposition

The Gallant-Lambert-Vanstone (GLV) method accelerates individual scalar multiplications kP on certain elliptic curves that have an efficiently computable endomorphism ϕ. An endomorphism is a map from the curve to itself. For GLV, we need a scalar λ such that $\phi(P) = \lambda P$ for all points P.

The scalar k is decomposed into two smaller scalars, k_1 and k_2, each approximately half the bit-length of k, such that:

$$k = k_1 + k_2\lambda \pmod{n}$$

where n is the order of the group. The scalar multiplication can then be rewritten as:

$$kP = (k_1 + k_2\lambda)P = k_1P + k_2(\lambda P) = k_1P + k_2\phi(P)$$

Since k_1 and k_2 are half the size of k, this computation can be performed much faster than the original kP, effectively halving the number of expensive point doublings. This decomposition can be applied to all scalars in the MSM before applying the Pippenger method, further reducing the overall cost.

4 References

- MSM Algorithm Note by drouyang
- Efficient Multi-Exponentiation by J. Bootle
- Improved techniques for multi-exponentiation
- zPrize Results Announcement