Article

Not peer-reviewed version

Optimizing Multi-Scalar Multiplication Over Fixed Bases

Saulius Grigaitis *

Posted Date: 2 April 2026

doi: 10.20944/preprints202604.0045.v1

Keywords: multi-scalar multiplication; elliptic curve cryptography; pippenger's algorithm; bucket method; precomputation; fixed base scalar multiplication

 doi:10.20944/preprints202604.0045.v1

Article

Optimizing Multi-Scalar Multiplication Over Fixed Bases

Saulius Grigaitis

Grandine, Lithuania; saulius@grandine.io

Abstract

This work investigates multi-scalar multiplication (MSM) over a fixed base for small input sizes, where classical large-scale optimizations are less effective. We propose a novel variant of the Pippenger-based bucket method that enhance performance by using additional precomputation. In particular, our approach extends the BGMW method by introducing structured precomputations of point combinations, enabling the replacement of multiple point additions with table lookups. We further generalize this idea through chunk-based precomputation, allowing flexible trade-offs between memory usage and runtime performance. Experimental results demonstrate that the proposed variants significantly outperform the Fixed Window method for small MSM instances, achieving up to 3× speedup under practical memory constraints. These results challenge the common assumption that bucket-based methods are inefficient for small MSMs.

Keywords: multi-scalar multiplication; elliptic curve cryptography; pippenger's algorithm; bucket method; precomputation; fixed base scalar multiplication

1. Introduction

Multi-scalar multiplication (MSM) is a fundamental operation in elliptic curve cryptography and underpins many cryptographic protocols. While MSM has been extensively studied, most recent advances focus on large-scale instances, where Pippenger's bucket method and its variants dominate in performance.

In contrast, small MSM instances exhibit different characteristics and allow alternative optimization strategies. In this regime, precomputation becomes a powerful tool, as its cost can be amortized over relatively small inputs. This opens the door to algorithms that trade memory for improved runtime efficiency.

However, classical bucket-based methods such as the BGMW approach are often considered suboptimal for small MSMs. In this work, we challenge this assumption by introducing novel precomputation technique that reduce the number of required group operations.

We propose a new variant that extends the BGMW method and demonstrate that, when combined with carefully designed precomputation strategies, bucket-based methods can outperform widely used approaches such as the Fixed Window method in the small MSM regime.

2. Previous Work

Multi-scalar multiplication (MSM) over fixed base has been extensively studied over several decades. Recent advances have primarily targeted large-scale MSM instances, typically of size at least 2^{20}, where state-of-the-art algorithms are based on variants of Pippenger's bucket method [1].

In contrast, smaller MSM instances admit different optimization strategies, most notably through the use of precomputation. In this regime, the precomputation cost is often proportional to the MSM size, allowing relatively modest overhead. Furthermore, smaller instances enable significantly larger precomputation per point, creating opportunities for more sophisticated algorithms that trade memory for improved runtime performance.

 doi:10.20944/preprints202604.0045.v1

This chapter reviews classical MSM algorithms.

2.1. Shamir's Trick

Shamir's Trick [2] computes a double-scalar multiplication $S = k_0P_0 + k_1P_1$ using a minimal precomputation table consisting of $\mathcal{O}$, P_0, P_1, and $P_0 + P_1$. Only the latter requires explicit precomputation.

The scalars are processed simultaneously in binary form from the most significant bit to the least significant bit. At each step, the accumulator is doubled, and a precomputed value is conditionally added based on the current bit pair. This method achieves an efficient trade-off between precomputation and runtime cost.

2.2. Straus's Method

Straus's method [3] generalizes Shamir's Trick to n scalars:

$$S = \sum_{i=0}^{n-1} k_i P_i.$$

The algorithm processes all scalars in parallel and relies on a precomputation table containing all possible sums of subsets of $\{P_0, \ldots, P_{n-1}\}$. At each bit position, the accumulator is doubled and the subset sum corresponding to active bits is added.

While computationally efficient—requiring $k - 1$ doublings and $k - 1$ additions for k-bit scalars—the method suffers from exponential precomputation complexity, rendering it practical only for very small n.

2.3. Bos-Coster Method

The Bos-Coster method [4] is a recursive MSM algorithm that iteratively reduces the largest scalars. At each step, the two largest scalars $k_0 > k_1$ are replaced as

$$k_0P_0 + k_1P_1 \;\rightarrow\; (k_0 - k_1)P_0 + k_1(P_0 + P_1).$$

Each iteration requires one scalar subtraction and one group addition. The process continues until a single scalar remains. The method effectively reduces scalar sizes while increasing point reuse, making it attractive in settings where additions are significantly cheaper than scalar operations.

2.4. Fixed Window Method

The Fixed Window method [5] accelerates scalar and multi-scalar multiplication through per-point precomputation. For a window size w, each point P_i is expanded into

$$\{P_i, 2P_i, \ldots, (2^w - 1)P_i\}.$$

Scalars are partitioned into w-bit chunks, and each chunk selects a precomputed multiple. The contributions are combined using appropriate doublings. The method achieves a predictable trade-off between precomputation cost $O(n2^w)$ and runtime efficiency.

2.5. Sliding Window Method

The Sliding Window method [5] improves upon the fixed window approach by dynamically selecting windows. Instead of fixed partitions, the algorithm scans the scalar representation and processes a window beginning with a non-zero bit.

This reduces the number of additions by skipping zero segments, making the method particularly effective for scalars with low Hamming weight.

 doi:10.20944/preprints202604.0045.v1

2.6. Pippenger's Bucket Method

For large-scale MSM, Pippenger's bucket method and its variants are widely regarded as the most efficient approaches [6]. The central idea is to group points according to scalar digits and accumulate them into buckets.

In the windowed variant, scalars are decomposed into base-2^w digits. For each window, points are assigned to $2^w - 1$ buckets based on their digit values. After bucket accumulation, the result is computed using a Horner-like scheme, followed by appropriate doublings between windows.

This approach significantly reduces the number of required scalar multiplications by replacing them with structured additions.

2.7. BGMW Method

The BGMW method [7] extends Pippenger's approach by incorporating structured precomputation. For each point P_i, a table of size h is constructed:

$$\{q^j P_i \mid i = 0, \ldots, n-1,\ j = 0, \ldots, h-1\},$$

resulting in $O(nh)$ precomputed points.

This moderate precomputation cost enables efficient bucket accumulation and improves data locality. However, a key characteristic of BGMW is the existence of an optimal parameter regime: increasing precomputation (e.g., by reducing the window size w) does not necessarily improve performance and even degrade it if w is below the optimal.

This behavior contrasts with fixed-window methods, where larger precomputation typically yields improvements. Although several extensions of BGMW have been proposed [6], they are not performant for scalar multiplication and small MSMs.

3. Pippenger's Bucket Method for Small MSMs

The bucket method improves performance by first aggregating points within each bucket and subsequently multiplying the accumulated sum by the corresponding bucket index. This strategy is particularly effective for large MSM instances, where the expected number of points per bucket grows with the MSM size n. In contrast, for small MSM instances, bucket occupancy is low, and the resulting performance gains are limited.

The BGMW method addresses this limitation by increasing the number of points per bucket through precomputation for Pippenger method. In particular, points corresponding to the same bucket index across different windows can be aggregated. This is enabled by the BGMW precomputation, which provides, for each point, its representation across all windows.

A straightforward attempt to further increase bucket occupancy would be to decrease the window size w. However, this approach is not effective, as the BGMW method admits an optimal window size. Reducing w below this optimum increases the total number of group operations and consequently degrades overall performance.

The main contribution of this work is the introduction of additional precomputation techniques for the Pippenger method, building upon the precomputations proposed in the BGMW method, which further improve performance for small MSM instances. In particular, the proposed variant outperform existing precomputation-based methods, such as the Fixed Window method, for small MSMs.

3.1. The Proposed Variant

The primary goal of the proposed method is to reduce the number of point additions required during bucket accumulation. For small MSM instances and moderate window sizes w: some buckets contain zero or one point, while other buckets contain multiple points. For such buckets with more than one point, point additions can be reduced by replacing explicit additions with lookups into precomputation tables containing sums of point pairs.

 doi:10.20944/preprints202604.0045.v1

More precisely, if a bucket contains two points, their sum can be obtained via a single lookup, provided that the corresponding pair has been precomputed. For buckets containing three points, two additions can be replaced by one lookup and one addition, and so on. In principle, one could precompute all possible point pairs; however, this becomes prohibitively expensive for large n or small w. To address this, we restrict precomputation to chunks of points. This significantly reduces memory requirements while preserving most of the performance benefits. The resulting trade-off is well suited for scenarios where memory usage is constrained.

The variant proceeds as follows. First, the standard BGMW precomputation of size nk/w is generated. Next, the resulting BGMW precomputation is used to precompute additional values of the form $P_i + P_j$ for indices

$$i, j \in \bigcup_{d=0}^{\frac{nk}{tw}-1} \{dt, dt+1, \ldots, (d+1)t-1\}, \quad i \neq j,$$

where w denotes the window size in bits, n is the MSM size, k is the scalar size in bits, and t is the chunk size.

The resulting algorithm follows the standard BGMW procedure, with modifications to the bucket accumulation phase. During accumulation, the algorithm scans each bucket and replaces pairs of points with their corresponding precomputed sums whenever possible. Increasing the chunk size t enlarges the set of available precomputed pairs, thereby reducing the number of explicit point additions.

4. Experiments

Authors of MSM algorithms often fork existing implementations, modify them, and publish the results. For example, [6] forked the `BLST` [8] library and released a modified version [9]. While this enables rapid prototyping, such implementations are typically tightly coupled to a specific elliptic curve cryptography (ECC) library, making fair comparisons across different libraries difficult. Moreover, these repositories are often not maintained, limiting reproducibility.

To address these issues, the author initiated the `Rust-KZG` project [10], a unified framework supporting multiple ECC backends, including `Arkworks` [11], `BLST` [8], `Constantine` [12], `MCL` [13], and `ZKCrypto` [14]. It integrates MSM implementations under a common interface, enabling fair and reproducible comparisons. The source code for the methods proposed in this work will be publicly released upon acceptance of the corresponding paper.

`Rust-KZG` is implemented in Rust, providing memory safety and performance comparable to C, and is already used by other researchers [15].

The proposed methods are compared against the Fixed Window method, specifically the optimized `wbits` implementation from `BLST`. Both the proposed variant and `wbits` use batched addition. Experiments are conducted on the widely used `BLS12_381` curve [16], using the same backend to ensure consistency.

All experiments are performed on a system with an AMD Ryzen 9 5950X CPU, 64 GB of RAM, and Ubuntu 24.04, using the `Criterion` benchmarking library.

4.1. Experiment Results

The performance of the Fixed Window method improves as the window size increases, unless further increases no longer reduce the number of scalar windows. However, each increase doubles the size of the precomputation table, leading to poor scalability with respect to memory.

In contrast, the proposed variant inherit from the BGMW method the property that an optimal window size exists. Deviating from this optimal value degrades performance, even if the precomputation size increases. Furthermore, the proposed variant introduce an additional parameter, the chunk

 doi:10.20944/preprints202604.0045.v1

size (a multiplier of n), resulting in an optimal combination of window size and chunk size for a given memory budget.

Due to the fundamentally different precomputation strategies, direct comparison with the Fixed Window method is not straightforward, as matching precomputation sizes is often not possible. Therefore, the proposed variant are compared using configurations with equal or smaller precomputation size than the Fixed Window method.

The experimental results for the proposed variant indicate that the proposed variant achieves up to almost 3X performance improvements over the Fixed Window method for small MSM sizes. In particular, the runtime is reduced by 63.05% for MSM of size 2^1, 47.32% for size 2^2, 32.31% for size 2^3, 24.44% for size 2^4, and 11.41% for size 2^5. For MSM of size 2^6, the proposed variant reaches very similar performance (only -0.77% difference) compared to the Fixed Window method.

It should be noted that these improvements are achieved under specific memory budgets for precomputation, as detailed in the corresponding table. The proposed variant begins to outperform the Fixed Window method only when a sufficient memory budget for precomputation is available. However, the required memory remains relatively modest: 18.43 KB for MSM of size 2^1, 48.96 KB for size 2^2, 176.26 KB for size 2^3, 665.86 KB for size 2^4, and 2.59 MB for size 2^5.

Furthermore, the proposed variant exhibits both globally optimal parameter settings for each MSM size and locally optimal settings for specific memory budgets. For example, for MSM of size 2^2, the globally optimal configuration uses a window size of 4 and a chunk size of 64, resulting in a precomputation table of approximately 3.16 MB. However, if the available memory is limited to 1 MB, the optimal configuration changes to a window size of 4 and a chunk size of 9. The chunks parameter values are divided by n in the tables below. All reported execution times are measured in milliseconds.

Table 1. MSM size 2^1 Fixed-Window vs. The Proposed Variant.

	Fixed Window		The Proposed Variant				Improvement
w	Precomp.	Time	w	chunks	Precomp.	Time	
7	12.29 KB	0.0912	7	1	10.66 KB	0.0925	-1.46%
8	24.58 KB	0.0861	4	1	18.43 KB	0.0472	45.14%
9	49.15 KB	0.0848	4	3	42.62 KB	0.0430	49.36%
10	98.30 KB	0.0816	3	5	89.76 KB	0.0368	54.90%
11	196.61 KB	0.0802	3	10	166.56 KB	0.0337	57.98%
12	393.22 KB	0.0792	3	19	300.96 KB	0.0306	61.32%
13	786.43 KB	0.0755	3	50	723.36 KB	0.0281	62.74%
14	1.57 MB	0.0761	3	50	723.36 KB	0.0281	63.05%
15	3.15 MB	0.0760	3	50	723.36 KB	0.0281	62.99%
16	6.29 MB	0.0709	3	50	723.36 KB	0.0281	60.36%

Table 2. MSM size 2^2 Fixed-Window vs. The Proposed Variant.

	Fixed Window		The Proposed Variant				Improvement
w	Precomp.	Time	w	chunks	Precomp.	Time	
8	49.15 KB	0.1136	5	1	48.96 KB	0.0789	30.57%
9	98.30 KB	0.1087	5	2	87.36 KB	0.0742	31.78%
10	196.61 KB	0.1036	4	3	158.21 KB	0.0661	36.17%
11	393.22 KB	0.1001	4	6	301.06 KB	0.0584	41.66%
12	786.43 KB	0.0975	4	9	448.51 KB	0.0524	46.26%
13	1.57 MB	0.0932	4	25	1.12 MB	0.0501	46.19%
14	3.15 MB	0.0924	4	37	1.62 MB	0.0488	47.19%
15	6.29 MB	0.0921	4	64	3.16 MB	0.0485	47.32%
16	12.58 MB	0.0841	4	64	3.16 MB	0.0485	42.31%
17	25.17 MB	0.0876	4	64	3.16 MB	0.0485	44.59%

 doi:10.20944/preprints202604.0045.v1

Table 3. MSM size 2^3 Fixed-Window vs. The Proposed Variant.

Fixed Window			The Proposed Variant				Improvement
w	Precomp.	Time	w	chunks	Precomp.	Time	
9	196.61 KB	0.1569	5	1	176.26 KB	0.1258	19.80%
10	393.22 KB	0.1459	5	2	329.86 KB	0.1171	19.75%
11	786.43 KB	0.1403	4	3	608.26 KB	0.1041	25.77%
12	1.57 MB	0.1358	4	6	1.18 MB	0.0924	31.92%
13	3.15 MB	0.1261	5	18	2.70 MB	0.0863	31.61%
14	6.29 MB	0.1234	4	30	5.60 MB	0.0835	32.31%
15	12.58 MB	0.1221	4	43	7.06 MB	0.0833	31.74%
16	25.17 MB	0.1118	4	43	7.06 MB	0.0833	25.48%
17	50.33 MB	0.1138	4	43	7.06 MB	0.0833	26.75%
18	100.66 MB	0.1105	4	43	7.06 MB	0.0833	24.59%

Table 4. MSM size 2^4 Fixed-Window vs. The Proposed Variant.

Fixed Window			The Proposed Variant				Improvement
w	Precomp.	Time	w	chunks	Precomp.	Time	
10	786.43 KB	0.2381	5	1	665.86 KB	0.2122	10.87%
11	1.57 MB	0.2252	5	2	1.28 MB	0.1890	16.07%
12	3.15 MB	0.2126	5	5	3.12 MB	0.1605	24.49%
13	6.29 MB	0.1987	5	10	6.20 MB	0.1521	23.46%
14	12.58 MB	0.1918	5	19	10.99 MB	0.1450	24.44%
15	25.17 MB	0.1851	5	43	23.55 MB	0.1409	23.91%
16	50.33 MB	0.1677	5	43	23.55 MB	0.1409	16.02%
17	100.66 MB	0.1696	5	43	23.55 MB	0.1409	16.95%
18	201.33 MB	0.1624	5	43	23.55 MB	0.1409	13.25%
19	402.65 MB	0.1536	5	43	23.55 MB	0.1409	8.29%

Table 5. MSM size 2^5 Fixed-Window vs. The Proposed Variant.

Fixed Window			The Proposed Variant				Improvement
w	Precomp.	Time	w	chunks	Precomp.	Time	
11	3.15 MB	0.3524	5	1	2.59 MB	0.3458	1.88%
12	6.29 MB	0.3262	5	2	5.04 MB	0.3006	7.84%
13	12.58 MB	0.3090	6	6	12.50 MB	0.2738	11.39%
14	25.17 MB	0.2880	6	12	23.71 MB	0.2590	10.05%
15	50.33 MB	0.2808	6	28	49.66 MB	0.2488	11.41%
16	100.66 MB	0.2542	6	43	90.95 MB	0.2484	2.29%
17	201.33 MB	0.2525	6	43	90.95 MB	0.2484	1.63%
18	402.65 MB	0.2408	6	43	90.95 MB	0.2484	-3.14%
19	805.31 MB	0.2273	6	43	90.95 MB	0.2484	-9.30%
20	1.61 GB	0.2156	6	43	90.95 MB	0.2484	-15.23%

 doi:10.20944/preprints202604.0045.v1

Table 6. MSM size 2^6 Fixed-Window vs. The Proposed Variant.

Fixed Window			The Proposed Variant				Improvement
w	Precomp.	Time	w	chunks	Precomp.	Time	
12	12.58 MB	0.5244	5	1	10.18 MB	0.5787	-10.36%
13	25.17 MB	0.4866	6	3	25.10 MB	0.4904	-0.77%
14	50.33 MB	0.4566	6	6	49.87 MB	0.4689	-2.69%
15	100.66 MB	0.4360	6	12	94.70 MB	0.4562	-4.64%
16	201.33 MB	0.3920	6	25	186.71 MB	0.4462	-13.82%
17	402.65 MB	0.3938	6	43	363.66 MB	0.4432	-12.54%
18	805.31 MB	0.3698	6	43	363.66 MB	0.4432	-19.84%
19	1.61 GB	0.3514	6	43	363.66 MB	0.4432	-26.12%
20	3.22 GB	0.3318	6	43	363.66 MB	0.4432	-33.58%
21	6.44 GB	0.3320	6	43	363.66 MB	0.4432	-33.51%

5. Conclusions

This work presented a novel variant of the Pippenger-based MSM method that enhance performance for small MSM instances. The proposed approach builds upon the BGMW framework and introduces additional structured precomputation, enabling the replacement of multiple point additions with efficient table lookups.

The experimental results show that the proposed variants consistently outperform the Fixed Window method for small MSM sizes, achieving up to approximately 3× speedup under realistic memory constraints. Importantly, these improvements are obtained without excessive memory requirements, making the approach practical.

The results also demonstrate that bucket-based methods, when augmented with carefully designed precomputation, are highly competitive even in regimes where they are traditionally considered inefficient.

6. Further Research

The author is currently working on several extensions of the proposed variant. First, the method can be generalized by increasing the number of precomputed combinations, including triples, quadruples, and higher-order groupings of points. While this may further improve runtime performance, it introduces additional memory and computational overhead during precomputation. Nevertheless, this approach yields performance gains for very small MSM instances and even for single-scalar multiplication, while still maintaining moderate memory requirements.

Second, additional strategies for increasing bucket occupancy, such as the use of Booth encoding and related techniques, further improve performance.

Third, alternative chunk layouts increase performance. For example, overlapping chunks could enable more group operations to be replaced by lookup operations.

Finally, hybrid approaches that combine multiple MSM techniques, including representations based on wNAF, may provide further performance improvements.

References

1. Pippenger, N. On the Evaluation of Powers and Related Problems. *SIAM Journal on Computing* **1976**, *9*, 230–250.
2. Möller, B. Algorithms for multi-exponentiation. In Proceedings of the International workshop on selected areas in cryptography. Springer, 2001, pp. 165–180.
3. Straus, E.G. Addition chains of vectors (problem 5125). *American Mathematical Monthly* **1964**, *70*, 16.
4. Bos, J.; Coster, M. Addition chain heuristics. In Proceedings of the Conference on the Theory and Application of Cryptology. Springer, 1989, pp. 400–407.
5. Hankerson, D.; Menezes, A.; Vanstone, S. *Guide to Elliptic Curve Cryptography*; Springer, 2004.

 doi:10.20944/preprints202604.0045.v1

6. Luo, G.; Fu, S.; Gong, G. Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2023**, pp. 358–380.
7. Brickell, E.F.; Gordon, D.M.; McCurley, K.S.; Wilson, D.B. Fast exponentiation with precomputation. In Proceedings of the Workshop on the Theory and Application of of Cryptographic Techniques. Springer, 1992, pp. 200–207.
8. Supranational. blst: Multilingual BLS12-381 signature library. https://github.com/supranational/blst, 2020–2026. Accessed: 2026-03-06.
9. LuoGuiwen. MSM_blst: Multi Scalar Multiplication over the BLS12-381 curve utilizing blst. https://github.com/LuoGuiwen/MSM_blst, 2023–2026. Accessed: 2026-03-06.
10. Grandine. rust-kzg: A Multi-Backend KZG and MSM Framework. https://github.com/grandinetech/rust-kzg, 2020–2025. Accessed 2026.
11. arkworks contributors. arkworks-rs: Rust ecosystem for zkSNARK programming. https://github.com/arkworks-rs, 2022–2026. Accessed: 2026-03-06.
12. mratsim. Constantine: High-performance cryptography stack for verifiable computation and blockchain protocols. https://github.com/mratsim/constantine, 2024–2026. Accessed: 2026-03-06.
13. Shigeo Mitsunari (herumi). MCL: A portable and fast pairing-based cryptography library. https://github.com/herumi/mcl, 2015–2026. Accessed: 2026-03-06.
14. zkcrypto contributors. zkcrypto: Rust ecosystem for zero-knowledge cryptography. https://github.com/zkcrypto, 2018–2026. Accessed: 2026-03-06.
15. Dziembowski, S.; Faust, S.; Kędzior, P.; Mielniczuk, M.; Mohanty, S.K.; Pietrzak, K. Beholder Signatures. *Cryptology ePrint Archive* **2025**.
16. Boneh, D.; et al. BLS Signatures, 2023. IETF CFRG draft.