Exploring Algorithms for Multi-Scalar Multiplication on MNT Curves: Benchmarking and Analysis

Aaneel Shalman Srazali[1]

Supervised by Dr. Philipp Jovanovic and Maria Corte-Real Santos

A project thesis submitted in partial fulfilment
of the requirements for the degree of
BSc in Computer Science
at
University College London.

April 26, 2024

[1]**Disclaimer:** This report is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This report dives into multi-scalar multiplication (MSM) algorithms within the field of elliptic curve cryptography. The operation's significance extends from bolstering digital signature algorithms like Elliptic Curve-Digital Signature Algorithm (EC-DSA), rooted in the hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP), to forming the computational crux of pairing-based trusted setups for zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs). While the efficiency of MSM algorithms on BLS curves are well-documented, a notable gap persists in literature regarding their performance on Miyaji-Nakabayashi-Takano (MNT) curves. With the help of Arkwork's implementation of MNT curves, this report aims to bridge that gap. We establish the the Pippenger bucket method as the stand-out algorithm for MSM on MNT curves, with this superiority becoming more pronounced for larger number of scalar-point pairs. However, we identify that the subsum accumulation phase in this method represents the bottle-neck and becomes inefficient for larger window sizes relative to the size of the problem. Our analysis highlights parallelism as the single most important optimisation, constrained solely by the available performance cores. Additionally, we demonstrate that signed integer decomposition yields enhancements for larger window sizes, while the novel subsum accumulation algorithm proposed by Guiwen Luo et al. mitigates the naïve approach to subsum accumulation in the Pippenger bucket method. We show this novel algorithm shows improvements that are positively correlated to the number of empty buckets and exhibits impressive robustness by incurring minimal additional costs even when parameter configurations are unfavourable. Finally, we demonstrate the interactions between these optimisations when implemented together.

Acknowledgements

I would like to convey my heartfelt appreciation to my supervisors, Dr. Philipp Jovanovic and Maria Corte-Real Santos for their invaluable guidance and insight throughout this project. Their expertise, kindness and constant willingness to help have instilled in me the drive to go the extra mile and passion for continuous learning in the field of cryptography. I would also like to acknowledge the academic and administrative staff at UCL who have made this learning experience as enriching and enjoyable as it was.

I extend my gratitude to my family, in particular my parents and my brother, whose everlasting love and support have made this achievement possible. To my girlfriend and friends, those whom I have known for long and have consistently been there for me, thank you. A brief mention to the open-source software, libraries and broader academic and cryptographic communities that have facilitated my research.

This journey has been made more meaningful thanks to each and every one of you.

Aaneel Shalman Srazali

Contents

Chapter 1

Introduction

In an era where privacy and security are at the forefront of digital communication, the impetus to enhance cryptographic protocols has never been stronger. One particular protocol that has shown immense promise is zk-SNARKs. zk-SNARKs are cryptographic proofs that enable one party (the prover) to demonstrate to another party (the verifier) that a specific computation was carried out correctly, without revealing any details about the computation itself or its inputs [1].

These proofs are succinct (meaning they are produced and verified quickly) and non-interactive (meaning the proof is completed in a single message without the need for interaction between the prover and verifier). They have gained significant traction in recent years for enabling parties to prove the validity of confidential information, such as financial transactions, serving as the backbone of decentralised applications and blockchain technologies like Ethereum [2] and Zcash [3].

1.1 What is multi-scalar multiplication?

The computational efficiency of zk-SNARKs hinges on the performance of the MSM operation on elliptic curves. This report endeavours to contribute to this field by benchmarking and analysing MSM algorithms on MNT curves [4]. This operation simply refers to process of simultaneously multiplying several points by scalars and summing the results. In the context of pairing-based trusted setups for zk-SNARKs, provers generate proofs through operations on multiple points within an elliptic curve group, while verifiers authenticate these proofs using several pairings within the verification equation. These

MSM operations command a significant portion (around 70% [5]) of the computational load.

Similarly, MSM is utilised in the signature generation phase of the EC-DSA, where multiple point-scalar multiplications are aggregated to form a signature, providing the security needed for most modern secure web connections [6]. This aggregation process is echoed in batch forgery identification, where MSM's capacity to concurrently verify numerous digital signatures or credentials significantly bolsters the efficiency of large-scale transaction systems [7]. Moreover, MSM is a crucial operation for session key computation in the Elliptic Curve Diffie-Helmann (ECDH) protocol [8], enabling secure communications over public channels.

1.2 What are MNT curves?

Within pairing-based cryptography, MNT curves are elliptic curves that are distinguished by their small embedding degree. This is in contrast to BLS curves which, while also used in pairing-based cryptography, typically feature higher embedding degrees. BLS curves are generally considered to support more efficient implementations of pairing operations and offer a higher security level with shorter parameter sizes.

However, practitioners in the field are interested in MNT curves as it enables the implementation of recursive SNARKs, which involves using zk-SNARKs to verify computations that include other zk-SNARK proofs. This unique capability is crucial to enhance the scalability and privacy features of blockchain technologies, making MNT curves particularly valuable for advanced cryptographic applications. We discuss the significance of these curves further in Section 2.3.

1.3 Research objectives

This report aims to benchmark and analyse the efficiency of MSM algorithms on MNT curves. This objective was pursued through a multifaceted research approach. This encompassed a deep dive on the cryptographic applications and the broad field of elliptic curve cryptography through a thorough literature review. This investigation then covered the theoretical foundations of pairing-friendly elliptic curves and its cryptographic utility.

Special attention was given to zero-knowledge proofs, by drawing on ZKLearning's online lecture series [9]. Crucially, this research culminated in an in-depth study of multiple MSM algorithms, delving deeply into their intricate workings, comparing and contrasting their methodologies, and exploring the foundational principles behind their designs. This comprehensive analysis of key algorithms is meticulously documented in this report.

On the implementation front, an online course on the Rust programming language provided by Barron Stone on Linkedin Learning [10] was undertaken. The choice of Rust was influenced by the built-in support for MNT curves provided by the Arkworks library [11]. Given the major challenge in implementation arose from the understanding of complexities of MSM algorithms, we dedicate Chapter 4 to a detailed examination of the logic and theoretical complexities of these algorithms and optimisation strategies. Through this approach, the project accomplishes several key objectives:

1. **Efficient Rust implementation:** We present an efficient Rust implementation of a host of MSM algorithms including the state-of-the-art Pippenger bucket method and optimisations to it on the MNT4-298 curve in Arkworks.

2. **Comprehensive performance analysis:** We provide a detailed evaluation on the performance of key MSM algorithms on MNT curves. This evaluation provides insights into the efficiency and scalability of MSM algorithms across three parameters known to significantly affect performance. We also pinpoint optimal parameter configurations for different algorithms in hopes of informing the practical implementation of cryptographic protocols.

3. **Benchmark establishment:** We address a notable gap in current literature by establishing a benchmark for evaluating MSM algorithms on MNT curves. This benchmark hopes to serve as a tool for future research, offering a comparative lens through which the efficacy of these algorithms on MNT curves can be assessed for recursive zk-SNARK applications.

4. **Comprehensive overview of existing algorithms:** An in-depth survey and theoretical exploration of key MSM algorithms is also provided. This analysis incorporates novel interpretation of these algorithms and hopes to serve as a valuable reference point for both current and future cryptographic research.

1.4 Structure

This report will be structured as follows. In Chapter 2, we introduce lesser known cryptographic primitives that serve as necessary technical background to grasp the findings of this report. Chapter 3 details literature review of related work before Chapter 4 dives into the historical development of MSM algorithms and optimisations. In Chapter 5, we presents the methodology behind our experiments, followed by an evaluation of its results in Chapter 6. Finally, we conclude and provide direction for future work in Chapter 7.

Chapter 2

Preliminaries

Before we tackle MSM algorithms on MNT curves, we recall some definitions and concepts of lesser known cryptographic primitives to provide the necessary technical background to contextualise the findings in this report. Alternatively, readers may proceed to Chapter 4 and refer back to this chapter when needed.

2.1 Elliptic curve cryptography

First introduced in the mid-1980s by Miller [12] and Koblitz [13] independently, elliptic curve cryptography (ECC) a branch of cryptography that employs the mathematical structure of elliptic curves over finite fields to construct secure and efficient cryptographic systems. ECC has become a cornerstone in the field due to its high level of security with relatively small key sizes, making it an attractive choice for many applications. To summarise what an *elliptic curve* is, we consider the following definition [14].

Definition 2.1 (Elliptic curve [14])**.** *An* ***elliptic curve*** *E, in its simplest form, is defined over finite field $\mathbb{F}_p$ by the Weierstrass equation*

$$y^2 = x^3 + ax + b \tag{2.1}$$

where p is a prime number, $a, b \in \mathbb{F}_p$ and $\Delta = 4a^3 + 27b^2 \neq 0$ (to avoid singular curves).

The points (x, y) on the curve, along with a distinguished point at infinity, form an abelian group under an addition operation. The non-singularity condition ensures the curve's smoothness, a critical property for defining a well-behaved group structure. Fig-

ure A.1 depicts an example of a canonical elliptic curve with $a = -1$ and $b = 1$, when defined over real numbers. The security of ECC relies on the difficulty of the ECDLP, which states that given two points $P, Q \in E(\mathbb{F}_p)$, finding scalar k such that $Q = k \cdot P$ is computationally unfeasible when p is large enough.

As stated in Section 1.1, ECC is utilised in various cryptographic protocols, most notably in zk-SNARKs, EC-DSA and the ECDH. In these protocols, MSM represents a fundamental operation and can be defined as the computation of

$$\sum_{i=1}^{n} k_i \cdot G_i = k_1 \cdot G_1 + k_2 \cdot G_2 + \ldots + k_n \cdot G_n \tag{2.2}$$

where k_i are scalars and G_i are points on E. The algorithms we analyse in this paper are essentially algorithms that try to optimisation the computation of the MSM solution when n is large. Therefore, efficient algorithms for MSM, that we will consider in this paper, are crucial for the practicality of ECC. We provide definitions and a and geometric visualisation of the basic elliptic curve operations via the chord-and-tangent rule in Section A.1.

It is crucial to note that the choice of elliptic curve parameters must be made carefully to avoid known vulnerabilities and to ensure a high level of security. The ECC community has established standard curves with tested parameters, such as the NIST curves and Curve25519, which have become widely adopted for their security and performance [15].

The appeal of ECC in practical cryptographic systems is not only theoretical but also computational. The time complexity of the best-known algorithms for the discrete log problem in ECC is proportional to the square root of the group's order, denoted as $O(\sqrt{r})$ [14]. Such an efficiency gain enables secure cryptographic operations with significantly smaller key sizes than those required by traditional systems like RSA and ElGamal [16], for similar levels of security. Specifically, for security equivalent to AES-128, an elliptic curve over a field size of approximately 2^{256}. is sufficient, while the traditional RSA would require a much larger key size, typically 2^{3072} [17].

This efficiency extends to the elliptic curve operations themselves. ECC's group operations over fields with a 256-bit prime base are far less computationally intensive compared to traditional RSA operations over a 2048-bit prime [14]. The smaller field sizes lead to

fewer arithmetic operations, resulting in faster and more responsive cryptographic applications. The practical benefits of ECC are therefore invaluable. In this report, the nuances of ECC will serve as the underpinning for more complex cryptographic constructions, such as MSM on elliptic curves.

2.2 Pairing-friendly elliptic curves

Pairing-friendly elliptic curves are a class of elliptic curves that facilitate the efficient computation of bilinear pairings, which are mappings that enable a direct algebraic relationship between two cryptographic groups. Bilinear pairings have revolutionised certain cryptographic applications by enabling advanced protocols that were not feasible with traditional elliptic curve operations alone. We formalise the concept of *pairing* through the following definition [14].

Definition 2.2 (Pairing [14])**.** *Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be three cyclic groups of prime order q where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. A **pairing** is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ satisfying the following properties:*

1. *bilinear: for all $u, u' \in \mathbb{G}_0$ and $v, v' \in \mathbb{G}_1$ we have*

$$e(u \cdot u', v) = e(u, v) \cdot e(u', v) \qquad \text{and} \qquad e(u, v \cdot v') = e(u, v) \cdot e(u', v) \tag{2.3}$$

2. *non-degenerate: $e(g_0, g_1)$ is a generator of $\mathbb{G}_T$*

Deriving from the definition above, a pairing-friendly elliptic curve is an elliptic curve over a finite field for which there exists a non-degenerate, efficiently computable bilinear pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$. The groups $\mathbb{G}_0$ and $\mathbb{G}_1$ are typically subgroups of the curve's rational points, chosen so that the discrete logarithm problem is hard in both groups and target group $\mathbb{G}_T$. The target group $\mathbb{G}_T$ is of great importance in pairing-based cryptography as on top of being large enough to make the discrete logarithm problem infeasible, it must be structured in a way that allows for the efficient computation of the pairing function.

The most commonly used pairings in cryptographic applications are the Weil pairing and the Tate pairing, along with their variants such as the Ate and optimal pairings, which offer better computational efficiency [18]. These pairings enable protocols such as Identity-Based

Encryption (IBE), Short Signatures, and Private Information Retrieval (PIR) schemes [19]. For example, the bilinear properties are pivotal in IBE systems to facilitate operations such as encryption and decryption using a public parameter and a unique identifier, without the need for certificates [19].

To construct pairing-friendly curves, cryptographers must balance several competing factors: the curves must have a sufficiently large order to prevent discrete log attacks, yet the order must have specific mathematical properties to enable efficient pairings. Curves such as Barreto-Naehrig (BN) and MNT curves are designed to achieve this balance [20].

The role of pairing-friendly curves is significant in zk-SNARKs, as they facilitate the creation of succinct and non-interactive proofs. For instance, the BLS12-381 curve, introduced in [21], has become a standard choice for many zk-SNARK-based systems due to its security level and efficiency in pairing computation. This curve specifically is also used in blockchain and privacy preserving cryptographic constructs [22], [23].

2.3 The significance of MNT curves

MNT curves are a family of elliptic curves named after Miyaji, Nakabayashi, and Takano, who first proposed them [4]. This seminal work offered explicit constructions for elliptic curves with specific fixed small embedding degrees, notably $k = 3, 4$, and 6. By solving particular Diophantine equations, the MNT methodology can generate curves with these predetermined embedding degrees, facilitating efficient pairings crucial in systems equating a 128-bit security [24].

The distinguishing features of MNT curves is indeed this small and fixed embedding degree, denoted as d, which measures the degree of the smallest field extension over which the group of r-torsion points (points P such that rP is the identity element) of an elliptic curve can be embedded [25]. The embedding degree directly impacts the efficiency of pairing computations: a smaller d typically leads to faster pairings, making MNT curves especially attractive for protocols that rely on pairings.

Definition 2.3 (Embedding degree [18])**.** *For a prime p that divides the number of points on an elliptic curve E over a finite field, the* ***embedding degree*** *d is the smallest positive*

integer such that d divides $(p^d - 1)$ *It corresponds to the field extension degree required to construct the finite field where the pairing's target group* $\mathbb{G}_T$ *resides.*

Due to their lower efficiency compared to BLS curves, MNT curves haven't garnered much attention. Despite this, no other curve can construct recursive SNARKs, which is why MNT curves are important. With that, we consider the definition of cycles of pairing-friendly elliptic curves, central to this paper, below [26].

Definition 2.4 (Cycles of elliptic curves [26])**.** *An* ***m-cycle of elliptic curves*** *is a list of m distinct elliptic curves* $E_1/\mathbb{F}_{p_1}, \ldots, E_m/\mathbb{F}_{p_m}$*, where* $_1, \ldots, p_m$ *are prime, such that the number of points on these curves satisfy*

$$\#E_1(\mathbb{F}_{p_1}) = p_2, \ldots, \#E_i(\mathbb{F}_{p_i}) = p_{i+1}, \ldots, \#E_m(\mathbb{F}_{p_m}) = p_1. \tag{2.4}$$

The specific construction of the cycle-friendly MNT4 and MNT6 curves allows a zk-SNARK proof on one curve to be verified as a computation on another curve. This property is crucial for enabling recursive SNARKs, as a proof on one curve can be used as input data to create a new proof on the second curve.

We would like to re-iterate that while MNT curves offer many benefits, their construction and parameter selection must be handled with precision to avoid potential vulnerabilities. Research continues to optimise these curves, seeking to extend their practicality and security in an ever-evolving cryptographic field. We discuss this in Section 3.1.

2.4 Cryptographic protocols involving MSM

While real-life applications of MSM algorithms on elliptic curves are beyond the scope of this project, we wish to illustrate more clearly how it is applied to cryptographic protocols, expanding on the examples provided in Section 1.1.

2.4.1 zk-SNARKs

As established in Section 1.2, zk-SNARKs allow one party, the prover, to demonstrate possession of certain information to another party, the verifier, without revealing the information itself or requiring interaction with the verifier. In many zk-SNARK constructions, particularly those utilising polynomial commitment schemes like the Kate-Zaverucha-

Goldberg (KZG) commitments which provides succinct polynomial commitments with fast verification, MSM plays a critical role.

The generation of a zk-SNARK proof using KZG commitments typically involves committing to a set of polynomials, evaluating them at a random point, and then aggregating the resulting evaluations into a succinct proof using MSM. This aggregation process is a critical step where the prover combines various scalar-point multiplications into a single elliptic curve point, representing the proof [27].

It is also crucial to note that all zk-SNARK constructions use MSM directly, but those that rely on KZG commitments, or similar polynomial-based commitments, heavily depend on efficient MSM implementations. Efficient MSM algorithms are thus essential for a practical zk-SNARK system, where the proof generation and verification times are crucial.

2.4.2 EC-DSA

EC-DSA is a variant of the Digital Signature Algorithm (DSA) adapted for use with elliptic curves. In EC-DSA, signing a message requires generating a random scalar, multiplying it with a fixed generator point on the curve to produce another point, and then performing further operations that include more scalar-point multiplications. The verification process involves checking whether a point generated from the signature matches the point generated from the message hash and public key, which also relies on MSM for efficiency [6], [28].

2.4.3 EC-DH

EC-DH is an elliptic curve adaptation of the Diffie-Hellman Key Exchange protocol, which enables two parties to establish a shared secret over an insecure channel. Each party computes a public value by performing scalar multiplication on a fixed generator point and then exchanging these values. The shared secret is derived by each party performing scalar multiplication on the received value using their private scalar [29]. MSM becomes relevant in scenarios where batch processing of key exchanges is required or when the protocol is extended for group key agreement, necessitating simultaneous scalar multiplications.

In summary, the efficiency of MSM directly influences the overall performance and practicality of these protocols, underscoring its significance in the field of cryptography.

Chapter 3

Literature review of related work

In this chapter, Section 3.1 builds on Section 2.3, examining the evolving landscape of pairing-friendly elliptic curves to contextualise the development and position of MNT curves within this progressive field. In Section 3.2, we survey recent advances in the efficiency of MSM algorithms to provide a yardstick upon which our findings can be compared.

3.1 Developments in pairing-friendly elliptic curves

The pursuit of optimal elliptic curves for cryptographic applications remains an ongoing area of research. Notable early contributions in this field were came with the proposal of MNT curves [4] which provided a basis for understanding the significance of embedding degrees in pairing computations [24].

In ensuing research, Ben-Sasson et al. introduced the concept of recursive proof compositions with two-cycle pairing-friendly MNT curves [30]. Karabina and Teske established the correspondence between the MNT6 and MNT4 families by demonstrating the existence of a prime-order elliptic curve $E/\mathbb{F}_{q^6}$ within the MNT6 family if and only if there exists a counterpart curve $E'/\mathbb{F}_{q^4}$ in the MNT4 family, with both curves having the same prime order p [24].

Although these innovations marked a milestone, they also highlight the complexities involved in realising such systems at the desired security levels. Particularly, both curves in such a cycle typically necessitate large base fields to achieve a 128-bit security level, which, for example, led the Coda protocol to implement curves of 753 bits [31].

Further analysis by Chiesa et al. has indicated inherent limitations in discovering other suitable cycles of curves, presenting a challenge to the scalability of this approach [26]. Building pairing-friendly curves is different (and less restricted) than building curves that are both pairing-friendly and cycle-friendly. Cycles of elliptic curves allow recursive proof composition, a feature crucial for zk-SNARKs.

MNT curves, while facilitating the construction of cycles, are limited in number due to their specific embedding degrees. Despite this limitation, MNT curves remain significant because they are a curve family that can be arranged into a cycle, with MNT4 and MNT6 curves for example. An alternative proposition, introduced by Bowe et al., was the Zexe system, which uses a BLS12 curve for inner proofs and constructs a second curve via the Cocks-Pinch method for outer proofs.

Addressing the need for an optimised pairing-friendly curve, recent advancements have shifted to the Brezing–Weng method for curve generation. A new curve generated using this method, proposed by El Housni and Guillevic [32], is more secure and faster than its predecessors, serving as a suitable substitute for the outer curve in Zexe's framework. Notably, this curve reduces the prime field size from 782 bits to 761 bits, leading to a significant performance gain. Although this achieves improved efficiency at the 128-bit security level, it cannot handle infinite recursion because it relies on chains rather than cycles of elliptic curves [33].

3.2 Advancement in MSM efficiency

This section provides an overview of the current state-of-the-art MSM algorithms within the domain of elliptic curve cryptography literature. A detailed exploration of the underlying principles, logic and complexities of key algorithms is presented in Chapter 4.

The Pippenger's bucket method emerges as a leading solution for large sets of scalar-point pairs [34]. By partitioning the computation into smaller segments, Pippenger's method optimises computational resource use and therefore, is commonly used in blockchain technologies. For scenarios where the number of scalar-point pairs is relatively low but the scalars k_i are large, Bos-Coster's algorithm offers an efficient alternative. This algorithm finds its niche in cryptographic applications where scalar precision is crucial, like secure

voting systems and encrypted messaging [35]. The simplicity and efficiency of the Straus algorithm, also known as Shamir's trick, make it suitable for environments with limited resources. By processing bits of scalar values simultaneously, it reduces computational complexity, making it ideal for lightweight cryptographic devices such as smart cards and IoT devices [36].

The research community has also explored vectorisation techniques to leverage modern CPU architectures, allowing parallel processing. This approach has shown promise in reducing the execution time of MSM operations, making it a topic of active research. A variant of 2-NAF decomposition in signed integer decomposition is also regularly discussed and said to improve the Pippenger bucket method in certain conditions [37]. A recent study by Guiwen Luo et al. introduced a novel MSM optimisation that combines the strengths of Pippenger's and Bos-Coster's algorithms, providing an adaptive approach that dynamically adjusts based on the input set's characteristics [38]. This report will analyse the algorithms and optimisations above in detail.

Shifting away from algorithmic headways, there have also been notable recent advancements in hardware-based optimisations. For instance, Shen et al. recently proposed a GPU-based optimisation on the Pippenger algorithm which brings about 1.01 to 1.12 in throughput under a suitable range of group sizes [5]. In similar vein, Aasaraai et al. with HardcamlMSM, propose FPGA-based acceleration on BLS12-377 which enhanced the computation of MSM with large N such as $N = 2^{26}$ to under 5.518 seconds [39], improving on CycloneMSM's runtime of 5.66 on the same parameters which in 2022, achieved more than a 4 times performance increase over existing software implementations [40]. Also earlier this year, Zhu et al. introduced a modular and adaptive parameter configuration technique, ElasticMSM, which claims to bring up to a significant 28x and 45x improvement on parallelised Pippenger bucket method by allowing for scalable optimisation of MSM tasks through preprocessing [41].

These recent hardware-based breakthroughs could significantly impact the field. As far as our knowledge goes, practical implementations and optimisations, are still under exploration. In this report, we focus on software-based improvements to MSM algorithms rather than hardware solutions. Nevertheless, these advancements underscore the ever-evolving nature of optimisation techniques and the continuous demand for improved efficiency in MSM algorithms.

Chapter 4

Multi-scalar multiplication algorithms

In this chapter, we analyse the logic and theoretical computational requirements behind various MSM algorithms on elliptic curves. Our exploration starts with foundational MSM algorithms (Section 4.2), advances to more sophisticated techniques (Section 4.3) and provides a comparative assessment (Section 4.4). Finally, we examine pivotal optimisations to the Pippenger bucket method (Section 4.5)

4.1 Problem statement

We begin with a formal definition of the problem of MSM on elliptic curves. Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$ (see Equation 2.1). Consider a set of n elements $G_1, \ldots, G_n$ in $\mathbb{G}$ where $|\mathbb{G}| = r$ and a set of n scalars $k_1, k_2, \ldots, k_n$ with b bits between 0 and $|\mathbb{G}|$. MSM on E is defined as the computation of

$$Q_{n,r} = Q_1 + Q_2 + \ldots + Q_n = \sum_{i=1}^{n} k_i \cdot G_i = k_1 \cdot G_1 + k_2 \cdot G_2 + \ldots + k_n \cdot G_n \tag{4.1}$$

where k_i are scalars and G_i are points on E. The challenge is to find an algorithm $f : (\mathbb{Z}, E(\mathbb{F}_p))^n \rightarrow E(\mathbb{F}_p)$ that optimises computational complexity while ensuring that no information about the scalars k_i or the elements G_i is leaked during computation.

4.2 Foundational MSM algorithms

For consistency, we will maintain the same notations as in Section 4.1 throughout this paper. We will also use A to represent the cost of point addition, $\lfloor x \rfloor$ to represent the smallest integer that is equal to or greater than x, $\lceil x \rceil$ to represent the largest integer that is equal to or lesser than x and $\mathcal{O}$ to represent the point at infinity or the identity element of the elliptic curve E. For simplicity of illustrating the computational complexities of the algorithms considered in this chapter, we will consider computational complexity from the perspective of number of point additions required, refer to point additions as additions and assume the cost of point doubling to also equal to A. Note that the cost of doubling, in practice, is less than the cost of addition, but it is the norm in ECC literature to consider them the same for ease of comparison and negligible difference.

4.2.1 Naïve MSM

A naïve approach to MSM would involve computing the sum of multiple scalar-point products $k_i G_i$ by performing scalar multiplication and summing the results to get the desired result $Q_{n,r} = \sum_{i=1}^{n} k_i \cdot G_i$. Considering the order r of group $\mathbb{G}$ and that scalars k_i are in the range $0 \leq k_i \leq r$, the worst-case computational complexity involves $(r-1)$ additions for each scalar-point products and a further $(n-1)$ additions to obtain $Q_{n,r}$. Overall, we obtain the worst case complexity to be:

$$[n \cdot (r-1) + (n-1)] \cdot A = [(n \cdot r) - 1] \cdot A \approx n \cdot r \cdot A \tag{4.2}$$

4.2.2 Trivial MSM

Next, we consider an approach commonly termed in ECC literature as trivial MSM. This approach takes advantage of the binary representation of each scalar k_i and looks at scalar multiplication as a series of doublings and additions. The double-and-add algorithm (algorithm Equation 4.2.2) used in trivial MSM exemplifies this.

The trivial MSM employs the double-and-add algorithm for each scalar-point pair (k_i, G_i) and then performs $(n-1)$ additions to obtain $Q_{n,r}$. The worst-case involves $(\log_2(r) - 1)$ doublings and, in the case of all bits being 1, up to $(\log_2(r) - 1)$ additions for each scalar

multiplication. Thus, the overall worst-case computational complexity of trivial MSM is

$$[2 \cdot (\lceil \log_2 r \rceil - 1) \cdot n + (n-1)] \cdot A = [2n \cdot \lceil \log_2 r \rceil - (n+1)] \cdot A \approx 2n \log_2 r \cdot A. \quad (4.3)$$

Algorithm 1 Double-and-add algorithm

Input: $k_i \in \mathbb{Z}$, $G_i \in E(\mathbb{F}_p)$
Output: $Q_i = k_i \cdot G_i$
$Q \leftarrow \mathcal{O}$ ▷ Initialise result
$P \leftarrow G_i$
while $k_i \neq 0$ **do**
    if $k_i \mod 2 = 1$ **then**
        $Q \leftarrow Q + P$ ▷ Add point to result
    $P \leftarrow 2 \cdot P$ ▷ Double point
    $k_i \leftarrow \lfloor k_i/2 \rfloor$ ▷ One right bit shift
return Q

4.2.3 Montgomery multiplication trick

The Montgomery multiplication trick, also referred to as the Montgomery ladder, is a constant-time algorithmic technique that avoids a key weakness of algorithm Equation 4.2.2 in that it is prone to timing-related side-channel attacks as its execution time depends on composition of 0 and 1 bits in the scalar. The "trick" to avoid this is to maintain a consistent sequence of operations by executing both an addition and a doubling operation at each iteration, making the computational path independent of the scalar's value. Thus, both the average and worst time complexity of the Montgomery Ladder would be equal to Equation 4.3.

The algorithm's validity relies on the invariant that, at each iteration, it maintains two accumulators, Q and P, such that after the i-th iteration, $Q = k_{j\ldots b} \cdot G$ and $P = (k_{j\ldots b}+1) \cdot G$, where $k_{j\ldots b}$ represents the processed bits of the scalar (or the portion of the scalar that have been accounted form up till step i) from position j to b. Therefore, if the j-th bit is 1, Q already represents the sum up to that bit, and P will be one addition ahead. If the bit is 0, the operations on Q and P effectively swap, maintaining the invariant. At the end, Q will hold the result of the scalar multiplication.

Algorithm 2 Montgomery ladder

Input: $k_i \in \mathbb{Z}$, $G_i \in E(\mathbb{F}_p)$
Output: $Q_i = k_i \cdot G_i$
$Q \leftarrow \mathcal{O}$
$P \leftarrow G_i$
for j from $b-1$ to 0 by -1 **do** ▷ k_i has b bits
    if $k_i[j] \mod 2 = 1$ **then** ▷ $k_i[j]$ is the j-th most significant bit of k
        $Q \leftarrow Q + P$
        $P \leftarrow 2 \cdot P$
    else
        $P \leftarrow P + Q$
        $Q \leftarrow 2 \cdot Q$
return Q

Despite the efficiency focus of this report, we wish to emphasise two aspects: first, the importance of securing against adversarial attacks and second, the principle of operation uniformity in this trick that we believe could be inspiration for future work on MSM algorithms (see Chapter 7).

4.3 Advanced MSM algorithms

This section delves into more sophisticated algorithms such as the Bos-Coster, Straus, and Pippenger bucket methods that leverage mathematical strategies to reduce computational requirements. By contrasting these algorithms, we wish to illustrate the progressive advancements in MSM techniques and justify our focus on the Pippenger bucket method for implementation based on its optimal performance in handling large batches of point-scalar multiplications. We also aim to highlight the key principles in these algorithms that have informed state-of-the-art algorithms and optimisations.

4.3.1 Bos-Coster method

The Bos-Coster algorithm [42] employs a mathematical trick that exploits the difference between the two largest scalars to reduce the number of required additions, a principle echoed in the optimisation we discuss in Subsection 4.5.3.

Through a priority queue mechanism, the process involves removing two tuples with the two largest scalars k_1 and k_2, computing $k_1 - k_2$ and P and G_2, and re-inserting $(k_1 - k_2, G_1)$

and $(k_2, G_1 + G_2)$ back into the queue. If $k_1 - k_2 = 0$, only the tuple $(k_2, G_1 + G_2)$ gets re-inserted, effectively reducing the number of tuples in the queue. This continues iteratively until only one tuple remains, this tuple is our desired result $Q_{n,r}$.

The correctness of this algorithm is guaranteed by the following equation.

$$[(k_1 - k_2) \cdot G_1] + [k_2 \cdot (G_1 + G_2)] = k_1 \cdot G_1 + k_2 \cdot G_2 \tag{4.4}$$

Note that sorting the tuples, while considered a computational overhead, is generally not the dominating factor.

Algorithm 3 Bos-Coster's algorithm

Input: $q = \{(k_i, G_i)\}_{i=1}^{n}$ where $\forall i, k_{i+1} \geq k_i$
Output: $Q_{n,r} = \sum_{i=1}^{n} k_i \cdot G_i$
$Q \leftarrow \mathcal{O}$
while len$(q) > 1$ **do**
    $k_1, G_1 \leftarrow q[0]$ ▷ Largest k
    $k_2, G_2 \leftarrow q[1]$ ▷ Second largest k
    q.pop$(q[0])$
    q.pop$(q[1])$
    $k_1 \leftarrow k_1 - k_2$
    $G_2 \leftarrow G_1 + G_2$
    if $k_1 > 0$ **then**
        q.append(k_1, G_1)
    q.append(k_2, G_2)
    q.sort() ▷ By k_i in descending order
return Q

Overall, this method provides a significant improvement in scenarios where the scalar values are large and varied. However, it degenerates to the trivial MSM in the worst case, when the difference between all consecutive scalars is 1. Thus, the worst case complexity is equal to Equation 4.3. However, in practice, this worst case scenario is typically unlikely and the average case complexity, while challenging to calculate precisely (as it depends on the scalar distribution), can be expected to be an improvement on trivial MSM.

4.3.2 Straus method

The Straus method [36], also referred to as Shamir's trick, is a blend of precomputation and an adapted variant of the the double-and-add algorithm we saw in (Equation 4.2.2).

Given a set of points G_i and corresponding scalars k_i for $i = 1, \ldots, n$, we first precompute $P_{i,m} = m \cdot G_i$ for all $0 \leq m < 2^c$ and $1 \leq i \leq n$, where c is the chosen window size. Next, we segment each b-bit scalar k_i into $\lceil \frac{b}{c} \rceil$ sections of c bits where $||$ represents concatenation: $k_i = k_{i,1}||k_{i,2}|| \ldots ||k_{i,\lceil \frac{b}{c} \rceil}$. We will refer to these segments as windows and segment values as window values. Then, for each i, we initialise a partial sum S_i to $P_{i,k_{i,1}}$, the precomputed point corresponding to the first window of the the scalar. Then, we consider the remaining concatenated window of k_i in ascending order, update S_i to be $2^c S_i + P_{i,k_{i,j}}$, where $P_{i,k_{i,j}}$ is the precomputed point $P_{i,m}$ for $m = k_{i,j}$ and $k_{i,j}$ the j-th window of scalar k_i of size c. Finally, we combine all partial sums through a variant of algorithm Equation 4.2.2 where we double and add regardless of the bit value to obtain our desired result $Q_{n,r} = \sum_{i=1}^{n} S_i$. We present this algorithm in the pseudocode below, which assumes the completion of all necessary precomputations.

Algorithm 4 Straus algorithm

Input: k_i, G_i, c
Output: $Q_i = k_i \cdot G_i$
$k_i \leftarrow k_{i,1}||k_{i,2}|| \ldots ||k_{i,\lceil \frac{b}{c} \rceil}$ ▷ Segment each scalar
$S_i \leftarrow P_{i,k_{i,1}}$
for j from 2 to $\lceil \frac{b}{c} \rceil$ by $+1$ **do**
    $S_i \leftarrow 2^c \cdot S_i$ ▷ Multiply S_i by 2^c
    $S_i \leftarrow S_i + P_{i,k_{i,j}}$ ▷ Add the precomputed value
return S_i

The algorithm above is then run for all $i = 1, \ldots, n$ with the same window size c before aggregating the partial sums to obtain $Q_{n,r} = \sum_{i=1}^{n} S_i$. The correctness of this algorithm is guaranteed by the following equation:

$$S_i = k_{i,1} \cdot G_i + 2^c \cdot k_{i,2} \cdot G_i + \ldots + 2^{\lceil \frac{b}{c} \rceil} \cdot k_{i,\lceil \frac{b}{c} \rceil} \cdot G_i = k_i \cdot G_i \tag{4.5}$$

where S_i in the equation above represents the product of the final iteration of doubling and adding for scalar k_i.

Since we need to precompute $P_{i,m} = m \cdot G_i$ for all $0 \leq m < 2^c$ and $1 \leq i \leq n$, and since for all i, $P_{i,0} = \mathcal{O}$ (which does not require precomputation), the overall storage requirements are

$$n(2^c - 1) \cdot G, \tag{4.6}$$

where G represents a single point storage cost.

Note that the cost of segmentation is considered to be negligible. Thus, in the worst case (where none of the scalar windows have value 0), we need to perform $(\frac{b}{c} - 1)$ additions (the first point is initialised) and $(\frac{b}{c} - 1) \cdot c$ doublings (last window does not require doubling) for each of the n scalar-point pairs. The worst case computational complexity is given by the following equation.

$$n \cdot [(\frac{b}{c} - 1) + c \cdot (\frac{b}{c} - 1)] \cdot A = (n + c + 1) \cdot (\frac{b}{c} - 1) \cdot A \approx (n + c) \cdot \frac{b}{c} \cdot A \tag{4.7}$$

The key principle in this algorithm is the segmentation of the binary representations of scalars to enable efficient accumulation and minimise the number of additions required. This principle lays the foundation for the state-of-the-art Pippenger bucket method.

4.3.3 Pippenger bucket method

The Pippenger bucket method (presented in [7], which is an application of the Pippenger algorithm in [34]) is widely recognised as state-of-the-art in the domain of MSM for its ability to optimise computations as the number of point-scalar pairs increases. This algorithm exemplifies an advanced paradigm in computational number theory, utilising a combinatorial framework to break down the MSM problem (Section 4.1) into smaller parts that can be solved independently and efficiently. This "divide and conquer" technique is pivotal in the field of zero-knowledge proofs, particularly zk-SNARKs, underpinning their scalability in blockchain technologies for enhanced transaction privacy and efficiency.

This algorithm can be broken down into three main steps. We will consider each of these steps in detail.

Step 1: First, this algorithm involves selecting a window size c and reducing the b-bit MSM into $\lceil \frac{b}{c} \rceil$ c-bit MSMs. This partitioning step involves a similar principle to the segmentation of scalars seen in line 6 in algorithm Section 4.3.2. The following visualisation

is provided by Gautam Botrel and Youssef El Housni in their paper titled *EdMSM: Multi-Scalar-Multiplication for SNARKs and Faster Montgomery multiplication* [37]. To maintain consistency with the notations across this report, we use $k_{i,j}$ to represent j-th window of scalar k_i and T_j the j-th c-bit MSM corresponding to $k_{i,j}$.

$$T_1 = [k_{1,1}] \cdot G_1 + \cdots + [k_{n,1}] \cdot G_n$$

$$\vdots$$

$$T_j = [k_{1,j}] \cdot G_1 + \cdots + [k_{n,j}] \cdot G_n$$

$$\vdots$$

$$T_{\lceil \frac{b}{c} \rceil} = [k_{1,\lceil \frac{b}{c} \rceil}] \cdot G_1 + \cdots + [k_{n,\lceil \frac{b}{c} \rceil}] \cdot G_n$$

Figure 4.1: Reducing the b-bit MSM into $\lceil \frac{b}{c} \rceil$ c-bit MSMs [37]

Step 2: In the second step, we solve each of the c-bit MSMs independently through a bucketing and subsum accumulation algorithm. Note that $\forall i, \forall j, 0 \leq k_{i,j} \leq (2^c - 1)$. Therefore, in the bucketing step, for each c-bit MSM T_j, $(2^c - 1)$ empty buckets are initialised and the points with window values m are accumulated into the bucket B_m for all $1 \leq m \leq (2^c - 1)$. This is pseudocoded in the algorithm below, which is repeated for all $1 \leq j \leq \lceil \frac{b}{c} \rceil$.

Algorithm 5 Bucketing step for the Pippenger method

Input: $T_j = [k_{1,j}]G_1 + \ldots + [k_{n,j}]G_n$
Output: $\{B_0, B_1, \ldots, B_{2^c-1}\}$
$B_m \leftarrow \mathcal{O}$ for all $m \in \{1, \ldots, 2^c - 1\}$ ▷ Initialise buckets
for i from 1 to n by +1 **do**
    $m \leftarrow k_{i,j}$
    $B_m \leftarrow B_m + G_i$
return $\{B_0, B_1, \ldots, B_{2^c-1}\}$

Recall that we have $\lceil \frac{b}{c} \rceil$ c-bit MSMs at the end of Step 1. Once the points for each c-bit MSM are bucketed, we employ what is called a subsum accumulation algorithm. For each set of buckets corresponding to the c-bit MSM, we iterate through the buckets B_m in descending order from $2^c - 1$ to 1 and iteratively add the accumulated point in the

bucket. The final result is the solution for c-bit MSM, $T_j = [2^c - 1]B_{2^c-1} + [2^c - 2]B_{2^c-2} + \cdots + [2]B_2 + [1]B_1$. In case the explanation above is challenging to comprehend, the visualisation below provided by Gautam Botrel and Youssef El Housni in [37]) presents the process intelligibly.

$$
\begin{array}{cccccccccc}
 & & B_{2^c-1} & & & & & & & \\
+ & & B_{2^c-1} & + & B_{2^c-2} & & & & & \\
+ & & \vdots & & \vdots & & & & & \\
+ & & B_{2^c-1} & + & B_{2^c-2} & + \cdots + & B_2 & & \\
+ & & B_{2^c-1} & + & B_{2^c-2} & + \cdots + & B_2 & + B_1 \\
\hline
= & [2^c - 1]B_{2^c-1} & + & [2^c - 2]B_{2^c-2} & + \cdots + & [2]B_2 & + [1]B_1
\end{array}
$$

Figure 4.2: Subsum accumulation process in the Pippenger method [37]

To make a like-for-like comparison with an optimisation we will study in Subsection 4.5.3, we also provide the subsum accumulation algorithm in pseudocode. Note that the process depicted by Figure 4.2 above and the algorithm below is repeated for each T_j for $1 \leq j \leq \lceil \frac{b}{c} \rceil$ to get the solution for each c-bit MSM.

Algorithm 6 Subsum accumulation step for the Pippenger method

Input: $\{B_1, B_2, \ldots, B_{2^c-1}\}$
Output: $T_j = [k_{1,j}]G_1 + \ldots + [k_{n,j}]G_n$
$P \leftarrow B_{2^c-1}$ ▷ Temporary point
$T_j \leftarrow B_{2^c-1}$ ▷ Initialise final result
for m from $(2^c - 2)$ to 1 by -1 **do**
    $P \leftarrow P + B_m$
    $T_j \leftarrow T_j + P$
return T_j ▷ Solution for j-th c-bit MSM

The validity of this step is guaranteed by the equation below [37] where the leftmost part of the equation $\sum_{i=1}^{m} i \cdot B_i$ is the target of the algorithm. The target is then broken into a double summation, where the inner sums each term B_i exactly i times and the outer sum takes care of the number of buckets. Finally, the equation reorders the terms of the double summation to group by each B_i. This change of the order of summation is valid due to the commutative property of addition and reflects the algorithm's process of accumulation.

$$\sum_{i=1}^{m} i \cdot B_i = \sum_{i=1}^{m} \sum_{j=1}^{i} B_i = \sum_{j=1}^{m} \sum_{i=j}^{m} B_i \tag{4.8}$$

Step 3: Now that we have the solutions for each c-bit MSM, the final step involves combining these to get our b-bit MSM solution, $Q_{n,r}$. This is done through the double-and-add variant we saw in the Straus method (lines 7 to 9 in algorithm Section 4.3.2). Since each b-bit MSM T_j corresponds to the j-th window of each scalar, this double-and-add variant effectively scales each window by 2^c j times. This ensures that every window is proportionally scaled to its position in the overall calculation. Note that in this report, we will consider the 1st window to have the most significant bits. Due to the difference in the input-output, we present the final part of the Pippenger bucket method in Equation 4.3.3.

Algorithm 7 Combination step for the Pippenger method

1: **Input:** $\{T_1, T_2, \ldots, T_{\lceil \frac{b}{c} \rceil}\}$
2: **Output:** $Q_{n,r}$
3: $Q_{n,r} \leftarrow T_1$
4: **for** j from 2 to $\lceil \frac{b}{c} \rceil$ by $+1$ **do**
5: $\quad Q_{n,r} \leftarrow 2^c \cdot Q_{n,r}$
6: $\quad Q_{n,r} \leftarrow Q_{n,r} + T_j$
7: **return** $Q_{n,r}$

The computational cost of the partitioning step (Figure 4.1) is generally considered to be negligible, this is corroborated by our findings in Chapter 6. For simplicity of calculating the computational complexity of solving each c-bit MSM, we will first consider this for each of the $\lceil \frac{b}{c} \rceil$ partitions. The cost of the bucketing phase of this step (algorithm Figure 4.3.3) equals to the expression below.

$$[n - (2^c - 1)] \cdot A \tag{4.9}$$

This can thought of as having to accumulate n points into $(2^c - 1)$ buckets and for each bucket, the cost of accumulating the first point in the bucket involves addition with $\mathcal{O}$ (which incurs negligible cost). Next, the computational complexity of the subsum accumulation phase is

$$2 \cdot (2^c - 2) \cdot A = [(2^{c+1} - 4)] \cdot A \tag{4.10}$$

where 2 additions are performed for each of the $(2^c - 2)$ iterations (as algorithm Section 4.3.3 elucidates). We would like to point out that this slightly disagrees with the $(2^{c+1} - 3)$

additions submitted in [37]. Finally, the complexity of the combining step (algorithm Equation 4.3.3) can be expressed as

$$(\frac{b}{c} - 1) \cdot (c + 1) \cdot A = [b - c + \frac{b}{c} - 1] \cdot A \tag{4.11}$$

as c doublings and 1 addition are executed for $(\frac{b}{c} - 1)$ iterations. Given the operations outlined for each stage, the overall computational complexity of the Pippenger bucket method is summarised by the following expression. From this expression, we can also gather that it is Step 2 that represents the computational bottleneck for this procedure.

$$[\frac{b}{c} \cdot (n + 2^c - 3) + (b - c + \frac{b}{c} - 1)] \cdot A \approx (n + 2^c) \cdot \frac{b}{c} \cdot A \tag{4.12}$$

4.4 Algorithm comparative analysis

The progression of MSM algorithms discussed in this report reflects a consistent drive towards efficiency. The following table summarises the theoretical worst-case computational complexities of the MSM algorithms discussed. Note that the average and worst-case complexities are identical for the Montgomery multiplication trick. Additionally, the ceiling operation on $\frac{b}{c}$ are omitted in the computational complexities of the Straus and Pippenger bucket methods $\lceil \frac{b}{c} \rceil$ to align with prominent ECC literature. This also reflects standard practice of selecting c such that it divides b evenly to reduce edge cases although selecting c such that $\frac{b}{c} \notin \mathbb{Z}$ does not compromise on correctness.

Algorithm	**Worst-case computational complexity**
Naïve	$n \cdot r \cdot A$
Trivial (Equation 4.2.2)	$2n \log_2 r \cdot A$
Montgomery ladder (Section 4.2.3)	$2n \log_2 r \cdot A$
Bos-Coster's method (Equation 4.3.1)	$2n \log_2 r \cdot A$
Straus method (Section 4.3.2)	$\approx (n + c) \cdot \frac{b}{c} \cdot A$
Pippenger bucket method (Equation 4.12)	$\approx (n + 2^c) \cdot \frac{b}{c} \cdot A$

Table 4.1: Worst-case computational complexities of MSM algorithms

The trivial approach to MSM refines this approach by using binary decomposition, which reduces the order of complexity by exploiting the binary representation of scalars. However,

the improvement plateaus when further scalability is required. The Montgomery multiplication trick, sharing the trivial method's complexity, distinguishes itself by its constant operational pattern, rendering it resilient to timing-related side-channel attacks.

Moving beyond these foundational algorithms, Bos-Coster introduces an element of subtraction to the additive landscape of MSM algorithms, iterating on pairs of the largest scalars, akin to a binary search. While its worst-case complexity remains bound by the trivial algorithm's framework, this worst-case is unlikely given that it requires $\forall i, j, \max(k_i - k_j) = 1$. Thus, it is likely to represent an improvement over its predecessors, particularly noticeable when scalars are sparse.

The Straus method extends upon this by incorporating a windowing technique into a variant of the double-and all algorithm. This method signifies a shift towards reducing the problem into windows, allowing for a reduction in computational steps and precomputation overhead. This scaling back of operations demonstrates a clear advancement over previous methods. However, it's important to note the trade-off in terms of storage requirements, as this method necessitates the precomputation and storage of $n \cdot (2^c - 1)$ points, which may impact the algorithm's applicability in resource-constrained environments.

The Pippenger bucket method culminates this developmental progress by implementing a sophisticated bucketing system that addresses the core inefficiency in previous algorithms: the repeated addition of the same points. By aggregating terms with similar window values prior to addition, Pippenger achieves a quasi-linear complexity that scales with the scalar size b and inversely with the window size c, presenting a method that adapts dynamically to the varying sizes of the inputs.

At first glance, the complexities in Table 4.1 may indicate that Pippenger's approach is less efficient than the Straus method. However, as presented by Guiwen Luo *et al.* [38], this isn't necessarily the case for large n and bigger c can be selected to minimise cost. Additionally, there are no precomputation requirements and the presence of the bucketing and subsum accumulation phases, absent in the Straus method, allows for additional optimisations. It is for these reasons that the Pippenger bucket method is generally considered to be state-of-the-art, hence our decision to center our implementation on this method, using the trivial MSM as a baseline for comparison.

4.5 Optimisations

An obvious but notable trend is that algorithms have sequentially built upon the insights and limitations of its predecessors. As stated in the section prior, the bucketing and subsum accumulation phases in the Pippenger bucket method make up the majority of the computational complexity and allows for additional optimisations. In this section, we aim to study these optimisations and analyse to what extent they improve on the state-of-the-art method.

4.5.1 Parallelism

Parallelism in computing is the simultaneous execution of multiple computations with the objective of improving computational speed. It leverages multiple processing units (cores) to perform various parts of a computation concurrently, rather than sequentially, and can be applied to algorithms that can be decomposed into independent tasks.

Applying parallelism to the Pippenger bucket method involves distributing the work of computing the buckets and performing subsum accumulation on them across multiple cores. This division allows for the concurrent solving of the c-bit MSMs, where each c-bit MSM is typically assigned to a separate core, depending on the specific implementation and the computational resources available. Recall that we have $\lceil \frac{b}{c} \rceil$ c-bit MSMs and the computational complexity of the Pippenger bucket method is $(n+2^c) \cdot \frac{b}{c} \cdot A$. Let y represent the number of cores, then the theoretical computational complexity of the Pippenger bucket method, when parallelised, can be represented as follows.

$$\begin{cases} \approx (n + 2^c) \cdot A & \text{if } \lceil \frac{b}{c} \rceil \leq y \\ \approx (n + 2^c) \cdot \lceil \frac{b}{c \cdot y} \rceil \cdot A & \text{if } \lceil \frac{b}{c} \rceil > y \end{cases} \tag{4.13}$$

The expressions above shed light on the main constraint of parallelism: the number of cores. If $\frac{b}{c} \leq y$, then the theoretical improvement on the Pippenger bucket method is by the factor of $\frac{b}{c}$. However, if the number of available cores continue to exceed $\frac{b}{c}$ further, the additional cores do not contribute to further speed-up due to the lack of independent partitions that can be worked on concurrently. Conversely if $\frac{b}{c} > y$, then this allows for the full utilisation of all cores, but some or all cores must handle multiple partitions sequentially. The theoretical improvement in this case is by the factor of $\lceil \frac{b}{c \cdot y} \rceil$, reflecting

the maximum number of partitions a core has to handle.

It is pivotal to note that the equations above assume an ideal scenario where tasks are evenly distributed with no overhead from parallelisation. However, in practice, this is often not the case. Firstly, the tasks of partitioning the b-bit MSM and combining the c-bit MSMs, albeit minimal, cannot be parallelised. Additionally, in practice, there exists variations in core performance due to factors like thermal design power, variable clock speeds, or other processes competing for computational resources which can lead to non-uniform execution times [43]. Finally, the movement of data between caches and processors in a multi-core system can add latency [44].

4.5.2 Signed Integer Decomposition

Next, we consider reducing the number of buckets by applying signed integer decomposition (SID) to the c-bit window values $1 \leq m \leq 2^c - 1$ in the $\lceil \frac{b}{c} \rceil$ partitions for each scalar. As submitted by Gautam Botrel and Youssef El Housni, the 2-Non-Adjacent Form (2-NAF) which encodes the scalar in signed binary form $\{-1, 0, 1\}$ is known to accelerate scalar multiplication $k_i \cdot G_i$ but does not help the Pippenger bucket method as the cost increases with the number of possible scalars regardless of their encodings [37].

However, an adapted form of 2-NAF in SID can be beneficial. This variant involves 2 main parts and the first part is conducted right after the partitioning step but before the bucketing step. In the first part, we represent the c-bit window values in the signed set $\{-2^{c-1}, \ldots, 0, \ldots, 2^{c-1} - 1\}$. This signed set representation can be achieved by iterating through the window values of a scalar $k_{i,j}$ in reverse order of significance (LSB to MSB or from partition $\frac{b}{c}$ to 1) and if the window value $k_{i,j} \geq 2^{c-1}$, we subtract 2^c from the window value $k_{i,j}$ (which forces it to be part of the signed set) and add 1 to the next window value. The validity of this step is ensured by the fact that 2^c in the current window is as significant as 1 in the next more significant window due to the doubling step in the combining phase which proportionally scales every window to its significance (see line 5 in algorithm Equation 4.3.3). This process is summarised in pseudocode in algorithm Section 4.5.2.

Note that the algorithm above can lead to overflow if the most significant window $k_{i,\frac{b}{c}} \geq 2^{c-1}$ so spare bits are initialised in the implementation to avoid this. The algorithm above

Algorithm 8 Signed integer decomposition

Input: $(k_{i,1}, \ldots, k_{i,\frac{b}{c}}) \in \{0, \ldots, 2^{c-1}\}$, c
Output: $(k'_{i,1}, \ldots, k'_{i,\frac{b}{c}}) \in \{-2^{c-1}, \ldots, 0, \ldots, 2^{c-1} - 1\}$
for j from $\lceil \frac{b}{c} \rceil$ to 1 by -1 **do**
    if $k_{i,j} \geq 2^{c-1}$ **then**
        $k'_{i,j} \leftarrow k_{i,j} - 2^c$
        $k_{i,j-1} \leftarrow k_{i,j-1} + 1$
    else
        $k'_{i,j} \leftarrow k_{i,j}$
return $\{k'_1, \ldots, k'_{\frac{b}{c}}\}$

is repeated for $1 \leq i \leq n$ before the second part of this optimisation which involves a slight adaptation to the bucketing algorithm (see algorithm Figure 4.3.3). Given the signed set, the computational saving is generated through the way the points G_i are accumulated in the buckets. Consider the new value of $k_{i,j}$ in the signed set. If $k_{i,j}$ is strictly positive, we add G_i normally into $B_{k_{i,j}}$. However, if $k_{i,j}$ is negative, we add $-G_i$ into the the bucket $B_{|k_{i,j}|}$, effectively reducing the number of buckets in (almost) half. For instance, if $c = 4$, instead of having to consider buckets $\{B_1, \ldots, B_7\}$, we now only have to consider buckets $\{B_1, \ldots, B_4\}$.

While this decomposition process does introduce additional computations, the cost is generally considered to be negligible [37]. This is supported by our findings in Chapter 6. Overall, SID's impact on computational complexity of the Pippenger bucket method is captured by the effect of bucket reduction. Firstly, it minimally increases the cost of bucketing from $\frac{b}{c} \cdot [n - (2^c - 1)] \cdot A$ to $\frac{b}{c} \cdot [n - (2^{c-1})] \cdot A$ as we now have $(2^{c-1} - 1)$ less additions with the point at infinity $\mathcal{O}$ and secondly, it reduces the cost of the subsum accumulation step from $\frac{b}{c} \cdot 2 \cdot (2^c - 2) \cdot A$ to $\frac{b}{c} \cdot 2 \cdot (2^{c-1} - 1) \cdot A$ as we now only have to iterate across 2^{c-1} buckets where the first bucket can be initialised and each iteration requires 2 additions (see algorithm Equation 4.5.3). Therefore, the theoretical computational complexity of the Pippenger bucket method with SID, assuming no overflow, is reduced from $\approx (n + 2^c) \cdot \frac{b}{c} \cdot A$ to the following expression.

$$\approx (n + 2^{c-1}) \cdot \frac{b}{c} \cdot A \tag{4.14}$$

4.5.3 More efficient subsum accumulation algorithm

The final optimisation we consider in this paper is a novel subsum accumulation algorithm proposed by Guiwen Luo et al. in their paper titled *Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs* [38]. This algorithm attempts to improve upon a potential inefficiency of the Pippenger bucket method during the subsum accumulation phase. This weakness manifests when there are empty buckets which is likely to occur when n is not significantly larger than c. While line 6 in algorithm Equation 4.5.3 can be considered cost-free as we add an empty bucket, line 7 involves an addition as long as we have encountered a non-empty bucket in a previous iteration.

This algorithm aims to address this issue by skipping over empty buckets that do not contribute value to the final sum through the subtraction on pairs of the largest values observed in the Bos-Coster method (see Subsection 4.3.1), leveraging on the potential sparsity of the buckets. As we only consider non-empty buckets, we need to introduce some notations. Let b_i refer to the window value of the i-th bucket, m the total number of non-empty buckets and let $b_0 = 0$. The first step is to initialise a temporary array `tmp` of size $d + 1$, where d is defined below.

$$d = \max_{1 \leq i \leq m} \{b_i - b_{i-1}\} \tag{4.15}$$

We then iterate through the non-zero buckets in descending order of i. In each iteration, the zeroth element of the *tmp* array, $tmp[0]$, is incremented by the value of the current bucket. This element essentially builds up a sum of all buckets, weighted by their respective scalar values. In the same iteration, we also calculate k, the difference between the current scalar b_i and the next scalar we iterate through b_{i-1}. If this difference $k \geq 1$, we update the k-th element in the `tmp` array by adding the accumulated sum from $tmp[0]$. This reflects the contribution of the scalar at the current index to a bucket corresponding to a larger scalar difference. After all iterations are complete, we perform the subsum accumulation algorithm Section 4.3.3 on the `tmp` array excluding `tmp`$[0]$ to produce the final sum T_j. The algorithm below summarises this process [38].

The correctness of this algorithm, as delineated in the paper by Guiwen et al. [38], is conceptualised by the transformation of the scalar multiplication sum into a series of accumulative computations. These computations exploit the differences between consecutive

Algorithm 9 Novel subsum accumulation algorithm [38]

1: **Input:** $\{b_1, b_2, \ldots, b_m\}$, $\{B_1, B_2, \ldots, B_m\}$
2: **Output:** $T_j = [b_1]B_1 + [b_2]B_2 + \ldots + [b_m]B_m$
3: Define a length-$(d+1)$ array $\texttt{tmp} = [0] \times (d+1)$
4: **for** $i = m$ to 1 **do**
5: $\quad \texttt{tmp}[0] = \texttt{tmp}[0] + B_i$
6: $\quad k = b_i - b_{i-1}$
7: $\quad$ **if** $k \geq 1$ **then**
8: $\quad\quad \texttt{tmp}[k] = \texttt{tmp}[k] + \texttt{tmp}[0]$
9: **return** $1 \cdot \texttt{tmp}[1] + 2 \cdot \texttt{tmp}[2] + \ldots + d \cdot \texttt{tmp}[d]$

scalars, captured by $\delta_j = b_j - b_{j-1}$. This transformation can be formally represented as follows [38].

$$\sum_{i=1}^{m}[b_i]B_i = \sum_{i=1}^{m}\left(\sum_{j=1}^{i}\delta_j\right)B_i = \sum_{j=1}^{m}\delta_j\left(\sum_{i=j}^{m}B_i\right) = \sum_{k=1}^{d}k\left(\sum_{\substack{j=1,\\ \delta_j=k}}^{m}\sum_{i=j}^{m}B_i\right). \tag{4.16}$$

The target of the algorithm $[b_i]B_i$ is expressed as the sum of scaled sums of the point B_i, where each scaled sum is weighted by δ_j. The sum is then reorganised to group terms by their corresponding δ_j values. The validity of this is ensured by the commutative and associative properties of addition, as in Equation 4.8.

When the sequence $\{b_i\}$ is strictly increasing (which is indefinitely the case in our implementation), the algorithm's cost is quantified as $2m + d - 3$ additions, with d representing $\max(\delta_j)$. When $d = 1$, this complexity of this algorithm degenerates to algorithm Section 4.3.3 [38]. Should the sequence $\{b_i\}$ not strictly increase, the algorithm optimises the computational path by omitting unnecessary updates to $\texttt{tmp}[k]$ when $k < 1$.

Overall, the heart of this optimisation lies in leveraging these differences δ_j, a clever mathematical trick which draws parallels to the Bos-Coster method. This insight's improvement on the relatively naïve approach to subsum accumulation seen in the Pippenger bucket method is positive correlated to the number of empty buckets. Even in the unlikely worst case where there are no empty buckets, the complexity is maintained with lines 5 and 8 in algorithm Equation 4.5.3 corresponding to lines 6 and 7 in algorithm Section 4.3.3.

Chapter 5

Methodology

The objectives of this report rests on a meticulous approach to implementing MSM algorithms on MNT curves. This chapter delineates the methodology applied to analysing the efficiency of these algorithms, from the infrastructure of the implementation (Section 5.1), the design considerations (Section 5.2) and finally, the experimental framework (Section 5.3). In general, the justification for each choice is anchored on efficiency; a critical aspect of this report.

5.1 Implementation infrastructure

The algorithms were all implemented and tested on a 2023 MacBook Pro equipped with the Apple M2 Pro chip. This model features a total of 10 cores, split between 6 performance cores and 4 efficiency cores, allowing for parallel processing of the Pippenger algorithm. The uniform testing environment ensures that our performance results are consistent and that comparisons between different algorithm implementations are fair and controlled.

A significant decision was the selection of Rust and the Arkworks suite of modules as the tool of choice for implementation. This decision was influenced by Rust's built-in support for MNT curves and its low-level language control, vital for the performance demands of this project. Furthermore, Rust's advanced compiler promotes secure memory management and thread safety, substantially mitigating the risk of runtime errors, a critical consideration for this report. Moreover, Rust's "zero-cost abstractions" allow it to run computations at speeds rivaling and sometimes surpassing C and C++, which are similarly renowned for efficiency but do not provide the same guarantees against memory-related errors.

Arkwork's modular design and specialised modules, including ark-ec for elliptic curve operations, ark-ff for finite field arithmetic, and ark-std for standard utilities like random number generation provide a robust foundation for implementing and testing MSM algorithms. The ark-mnt4-298 module was specifically used for its optimised implementations of MNT4-298 curves, aligning with the project's goal to test MSM on MNT curves.

The project, whose repository link to which can be found in Appendix E, is structured with a focus on modularity and clarity. At the root level, the Cargo.toml file sets forth the dependencies. The source folder (src) encapsulates the implementation of algorithms (naive.rs, pippenger.rs, etc.) and the operations required (operations.rs). The tests folder (tests) mirrors the src structure, ensuring each algorithm is accompanied by its test suite (naive_test.rs, pippenger_test.rs etc.).

5.2 Design considerations

To thoroughly assess the algorithms' scalability and efficiency, certain factors were fixed to maintain uniformity across different parameters. This section explains the rationale behind such fixed variables (elliptic curve, coordinate system and the exclusion of precomputation) and highlights some of the interesting programming choices made during implementation.

The MNT4-298 curve, characterised by a 298-bit prime and an embedding degree of k=4, was selected over other MNT variants in the Arkworks library due to its computational efficiency. The smaller finite field and reduced embedding degree assist in this regard, relative to other variants like MNT6-298 and MNT4-753m without affecting teh evaluation of performance of MSM algorithms. It is imperative to note, however, that in practical applications, this curve presents a lower security level and should be avoided [45]. Our selection is strictly for analytical purposes and not for deployment in security-critical environments.

Within Arkworks' implementation, $\mathbb{G}_1$ and $\mathbb{G}_2$ denote the two prime-order subgroups associated with the MNT4-298 curve. The choice of $\mathbb{G}_1$ over $\mathbb{G}_2$ again lies in $\mathbb{G}_1$'s computational efficiency as the subgroup is smaller and requires fewer resources. Furthermore, projective coordinates represented as $\{(X, Y, Z) = (\lambda x, \lambda y, \lambda),\ \lambda \neq 0\}$ defined over the

Weierstrass equation are favored over affine coordinates due to their elimination of costly field inversions, optimising the group law operations without affecting the curve's algebraic structure. We provide descriptions of alternative coordinate systems and equations of elliptic curves in Section A.2 and Section A.3 respectively.

Moreover, since the Pippenger bucket method includes three phases, a suitable data structure is required for the storage of window values and bit index of scalar paritions. In our implementation, we utilise BTreeMap for its sorted property, which is necessary when performing subsum accumulation as it relies on traversing scalar values in a sorted manner. This choice contrasts with HashMap, which, although generally faster for insertion and lookup, does not maintain order and thus would have required additional sorting steps.

For storing window values, a vector is chosen for its contiguity in memory and cache-friendliness, which is vital for the numerous iterative operations in the MSM algorithms. While other data structures like LinkedList offer efficient insertions and deletions, their non-contiguous storage could lead to performance hits due to cache misses, which is a crucial consideration for algorithms where iteration over elements is frequent.

Finally, there are no precomputation requirements for any of our implemented algorithms. Despite being a cornerstone in ECC literature, precomputation was excluded to purely evaluate the computational efficiency of the MSM algorithms. Moreover, storage becomes a critical limitation when performing scalability analysis.

5.3 Experimental framework

To comprehensively evaluate the efficiency of various MSM algorithms, an empirical approach was adopted. This section outlines the experimental framework employed, detailing the array of algorithmic implementations, the testing strategy, and performance metrics. This framework is designed to ensure a robust and rigorous comparison of MSM algorithms.

Our experimental suite encompasses ten different MSM algorithm implementations. This suite begins with the naïve approach to MSM which was implemented primarily as a benchmark, offering a means to verify the correctness of subsequent implementations. The trivial MSM algorithm was chosen to establish baselines for performance comparison. The state-of-the-art Pippenger bucket method was chosen as the focal algorithm due to its ability to

optimise computations as the MSM problem increases in size, crucial for zk-SNARKs. The three key optimisations to the Pippenger bucket method were all implemented, encompassing parallelism, signed integer decomposition to a novel subsum accumulation algorithm [38]. Finally, we implemented all linear combinations of these optimisations to explore their interactions comprehensively and identify the most efficient algorithm configuration. This ensemble of algorithms provides a broad spectrum for assessing performance across a variety of computational contexts.

To guarantee the correctness of each implementation, our testing suite compares the results of each algorithm against the naïve MSM output across wide ranges of number of point-scalar pairs. Our testing suite also encompasses unit tests for individual components within the algorithms and integrated tests that simulate real-world scenarios where points and scalars are randomly generated. In total, our test suite contains 142 tests, with 6 tests for the naïve and trivial implementations and at least 14 each for the remaining 8 algorithms, with this number varying with the optimisation.

Once correctness was verified, we assessed algorithm efficiency by utilising runtime as the performance metric, chosen as a direct efficiency indicator. To evaluate scalability and the impact of changing parameters, we recorded runtimes across a range of key variables critical to cryptographic research and known to significantly affect performance and applicability. These variables are listed below. Note that when testing for the impact of a specific variable, the others are held constant.

1. **Window size c:** Selected as a variable to understand its influence on the efficiency of the Pippenger algorithm and its optimisations as well as identify its optimal value for practical applications. $n = 10^3$ was fixed to reduce variance and $b = 32$ was chosen to enable a larger range of c to be tested.

2. **Number of point-scalar pairs n:** Varying the size of the problem allows us to study the algorithms' scalability, thereby informing the suitability of each MSM algorithm for various application sizes. The values are varied across a span multiple orders of magnitude, from 10 to 10^7, increasing multiplicatively by a factor of 10. $c = 8$ were chosen as it was determined to be the optimal window size for $b = 32$ and our hardware from prior testing.

3. **Scalar size 2^b:** Scalar size is intrinsically linked to security parameters in cryptographic systems. By evaluating a range of scalar sizes, we gain insight into how the bit-length of keys, which is a proxy for security level, impacts computation time. Scalar sizes we test range from 2^{10} to 2^{32}, increasing multiplicatively by a factor of 2^2 to assess algorithm performance across scalar ranges. $n = 10^3$ is fixed to reduce variance and $c = 2$ is chosen as it divides b evenly across these scalar sizes.

By holding certain variables constant during testing, we seek to identify configurations that minimise runtime while maintaining computational accuracy to provide valuable insights into the practical implementation of cryptographic protocols.

To align empirical findings with theoretical predictions, runtime measurements are also recorded for the distinct computational stages of the Pippenger method, as well as the SID optimisation. This segmentation serves to pinpoint performance bottlenecks and the additional computations introduced by SID to validate whether empirical data corroborate the computational complexity discussed in Chapter 4.

Each configuration is run ten times to account for variability in execution time due to factors like CPU cache state, memory allocation, and system states as well as to provide more reliable results. The mean ($\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$) of these runs represents the focal data point, while the standard error ($SE = \frac{\sigma}{\sqrt{n}}$) provides insights into the reliability of the results. For a normally distributed dataset, approximately 95% of the data points are expected to fall within two SE of the mean ($\mu \pm 2 \cdot SE$). The modified coefficient of variation ($CV = \frac{\sum_{i=1}^{n} |x_i - \mu|}{n \cdot \mu} \times 100$) is calculated to emphasise the average deviation of runtimes from the mean, expressed as a portion of the mean. This measure provides a proxy of the variability in relation to the average runtime. Finally, we also compute the normalised peak deviation ($NPD = \frac{\max(x_i) - \mu}{\mu}$) to measure the extent to which the worst-case complexity diverges from the average-case complexity, expressed as a percentage of the mean. It is important to note that the sample size for each statistical measure is 10, which may introduce variability and should be considered when interpreting the results.

This experimental framework lays the groundwork for the subsequent evaluation of results. It was crafted with attention to the principles of reproducibility, reliability and thoroughness, ensuring that the performance analysis conducted is both comprehensive and reflective of the algorithms' real-world applicability.

Chapter 6

Evaluation of results

This chapter aims to critically evaluate and compare the performance of MSM algorithms across a range of key independent variables. We start by considering the algorithms without optimisations in Section 6.1 before we thoroughly examine the effects of optimisations on the Pipenger bucket method in Section 6.2. Finally, we perform regression analysis to visualise the effects of key parameters to the best performing algorithms in Section 6.3. A complete listing of results analysed in this chapter is provided in Appendix B.

6.1 Algorithmic efficiency analysis

First, we note that our naïve implementation refers to the process of direct scalar multiplication, not the naïve approach of repeated point additions discussed in Section 4.2.

Naïve MSM vs. Trivial MSM: Between these two algorithms, we observe marginal difference across all parameters. Over 1000 points and varying scalar sizes, trivial MSM outperformed naïve MSM by a mere 0.3% on average. However, the slight outperformance can be observed just over 81% of all configurations. Therefore, this suggests to us that manual optimisation of scalar multiplication through explicit coding of point addition and doubling may yield more efficient outcomes than relying solely on compiler optimisations.

Trivial MSM vs Pippenger bucket method: The Pippenger bucket method consistently outperforms the trivial MSM across a spectrum of window sizes and scalar bit sizes. When operating with a window size of 2, Pippenger's method exhibits a substantial and stable performance gain, ranging between $64 - 66\%$ compared to trivial MSM, as scalar

sizes scale from 2^{10} to 2^{32}. Both methods show a predictable deceleration with increasing scalar sizes, aligning with our theoretical analysis presented in Chapter 4, where we established that the complexity of Pippenger's method scales with the scalar bit size b (Equation 4.12), and trivial MSM scales with the the maximum value of a scalar r (see Equation 4.3). The consistency in relative performance advantage, though, is an interesting observation. It suggests that there may be a constant value of around 2/3 that describes this improvement across all scalar sizes.

As for window sizes, with fixed values $n = 1000$ and $b = 32$, trivial MSM algorithm's performance is invariant as this parameter does not play a role in its computation. On the other hand, Pippenger's method unveils its proficiency with increasing window sizes, reaching peak improvement of 86% over trivial MSM at a window size of 8. This peak correlates with the critical point where the $2^c < n$. This is because as 2^c surpasses n, although the number of buckets are 2^{c-1}, which is less than 1000 at $c = 10$, the likelihood of a greater number of empty buckets increases as c increases and the relatively naïve approach to subsum accumulation becomes costly. Beyond $c = 8$, Pippenger's advantage wanes and at $c = 12$, it offers only a 67% improvement over trivial MSM. At $c = 16$, we have 32768 buckets for each partition and many unnecessary additions which led to a 271% deterioration on trivial MSM.

As we increase the number of point-scalar pairs, a trend emerges that underlines the scalability of the Pippenger bucket method. This is crucial for pairing-based zk-SNARK applications. Pippenger's method shows an improvement on trivial MSM by 59% at $n = 10^2$, 85% at $n = 10^3$ and reaches a peak performance improvement of approximately 88% for all n from 10^4 to 10^7. We note a common trend among all algorithms, including optimisations: Beyond 10^4, the runtime increase proportionally with n, by a factor of 10. The performance improvements are therefore, stable beyond 10^4.

To analyse the bottleneck between the three phases of the Pippenger bucket method alone, we present table showing the percentage of runtime taken by each steps Table 6.1 for $c = 8$, $n = 1000$ and $b = 32$. The results agree with our theoretical analysis that Step 2 represents the bottleneck, although it is interesting to observe the 30x difference between bucketing and subsum accumulation.

Steps	μ (ms)	**Percentage (%)**
Partition	0.339	0.708
Bucketing	1.516	3.168
Subsum	45.848	95.827
Combining	0.142	0.297
Total	47.844	100.000

Table 6.1: Pippenger bucket method bottleneck analysis ($c = 8$, $n = 1000$, $b = 32$)

6.2 Optimisation effects

Parallelism: First, we note that the improvement with parallelism should theoretically be as a factor of the number of partitions $\lceil \frac{b}{c} \rceil$ but as stated in Subsection 4.5.1, this assumes several factors and is limited by the number of cores. Generally, the results for parallelism are as expected.

It exhibits significant improvements over the Pippenger bucket method, between 74% and 81.25% for varying scalar sizes and between 70% to 74% for varying number of point-scalar pairs. It achieves a more than 4x increase when $b = 12$ and $b \geq 26$. When varying window size, we witness an impressive peak performance of $10.59ms$ at $c = 6$ and a peak improvement on the Pippenger method of 83% at $c = 2$. When $c = 6$, five c-bit MSMs and one 2-bit MSM is processed concurrently by our 6 performance cores whereas when $c = 26$, each performance core processes two 2-bit MSMs and each efficiency core one 2-bit MSM.

These results unfortunately inform us that the 6 performance cores are more performance-intensive than the 4 efficiency cores in our hardware configuration, which is a key consideration when interpreting results. The results also suggest that when number of cores is not a limiting factor, the probability of achieving theoretical improvements by factor $\lceil \frac{b}{c} \rceil$ decreases as the number of partitions increases. We believe this can be explained by the fact that with more partitions, it is less likely that the computation times are evenly distributed. Otherwise, the improvements by parallelism are as expected.

Signed integer decomposition (SID): First, we note for $n = 1000$, $c = 8$ and $b = 32$, the additional cost introduced by SID is on average $0.128ms$, 0.28% of the total runtime for the Pippenger bucket method with SID. This supports our hypothesis that it introduces negligible cost.

The overall runtime results suggest a strong positive correlation with window size. For $c = 2$, we notice a deterioration on the Pippenger bucket method for all scalar sizes and for $n \geq 10^4$. The deterioration does, however, decrease as scalar size increases from -19% at $b = 10$ to -5% at $b = 32$. This can be explained by the minimal effects of SID on low window sizes but effects that add up as scalar size increases. For instance, when $c = 2$, SID only reduces the number of buckets from 3 $(2^c - 1)$ to 2 (2^{c-1}) for each partition with the possibility of having introducing extra partition to handle overflow. Therefore, as window size increases, we do see SID provide more and more significant improvement. This is presented in Table 6.2 which shows the improvement over Pippenger for different window size. We exclude data for $c = 16$ here as both these algorithms exhibit outliers, performing worse than trivial MSM. The number of empty buckets and unnecessary additions in subsum accumulation are, however, significantly reduced by SID, but only by a factor of how many buckets it reduces which is approximately 2x.

Window size	2	3	4	6	8	10	12
Improvement (%)	-5.45	-3.94	-6.97	0.39	0.20	20.55	35.45

Table 6.2: Improvement of SID over Pippenger for $2 \leq c \leq 12$ ($n = 1000$, $b = 32$)

More efficient subsum accumulation [38]: At the end of Subsection 4.5.3, we theorised that this optimisation should be strongly positively correlated with the number of empty buckets but maintained in the worst case. Note that the number of empty buckets are effectively a function of n and c. The smaller the number of point-scalar pairs and the larger the window size, the higher the likelihood of having empty buckets.

For a fixed $c = 2$ and $n = 1000$, the optimised subsum accumulation [38] shows minor improvements of between 0.5% to 2.7% across all scalar sizes, with no clear trend as scalar size increases. This can be explained by the fact that when $c = 2$, there are only 3 buckets for each partition and the expected number of empty buckets are low. For $n = 10$ and $n = 10^2$, we see significant improvements of 69% and 43% respectively, supporting the hypothesis of positive correlation with the number of empty buckets. For fixed $c = 2$ and as the points scale to beyond 10^4, it exhibits a minimal deterioration of around 1.5%. This deterioration can be explained by the marginal additional cost from scanning for the two largest non-empty buckets and performing scalar subtraction being non-negligible for n large enough.

Most notably, however, this optimisation exhibits impressive improvements for large window sizes, clocking $45ms$ for $c = 16$, $n = 1000$ and $b = 32$, which represents a 96% improvement on the Pippenger bucket method. This observation signifies the importance of this optimisation as none of the other two optimisations were able to overcome the poor performance due to the weaknesses of a naïve subsum accumulation, most pronounced at $c = 16$. Therefore, although this optimisation causes marginal increase in cost for large n, it provides us a robust alternative if a big window size was to be preferred.

As for combinations of optimisations, we note that all optimisations, given they tackle different sections of the Pippenger bucket method, can be implemented together. The observed data indicates that, generally, these combinations result in the cumulative benefits of both individual optimisations. This outcome aligns with expectations. However, an interesting deviation is noted with the SID and novel subsum accumulation [38] combination, where the optimisations counter-act against each other. This interaction can be rationalised by considering that SID exhibits increased effectiveness with larger window sizes as it works by reducing the number of required buckets. However, a reduction in buckets reduces expected number of empty buckets which then undermines the effectiveness of this novel subsum accumulation algorithm.

6.3 Regression analysis

In this section, we apply quadratic regression to model and better visualise the non-linear relationship between key parameters and runtimes. We model for the two best performing algorithms in our experimental suite in the parallelised Pippenger (we will refer to this as Parallel) and the Pippenger with all 3 optimisations combined (we will refer to this as Parallel SID Subsum) as well as the Pippenger bucket method as a baseline for performance comparisons.

The quadratic regression was conducted by organising the data for the independent variable x into a design matrix $\mathbf{X}$, which includes the squared terms of x, the linear terms of x, and a column of ones for the intercept term. The observed outcomes were assembled into a response vector $\mathbf{Y}$. The coefficient vector β was then determined using the least squares method, calculated as $\beta = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}$, which minimises the sum of squared differences between the observed and predicted values. We note that the model is subject

to inherent model bias and the disproportionate influence of highly deviated data points. Nevertheless, it is a simple enough model that illustrates clearly the variations across different parameters. In analysing performance with varying window sizes, we omitted $c = 16$ as the parallelised Pippenger and Pippenger algorithms exhibit outliers for this value. We have also opted to exclude the model for number of point-scalar pairs due to a low r^2 value.

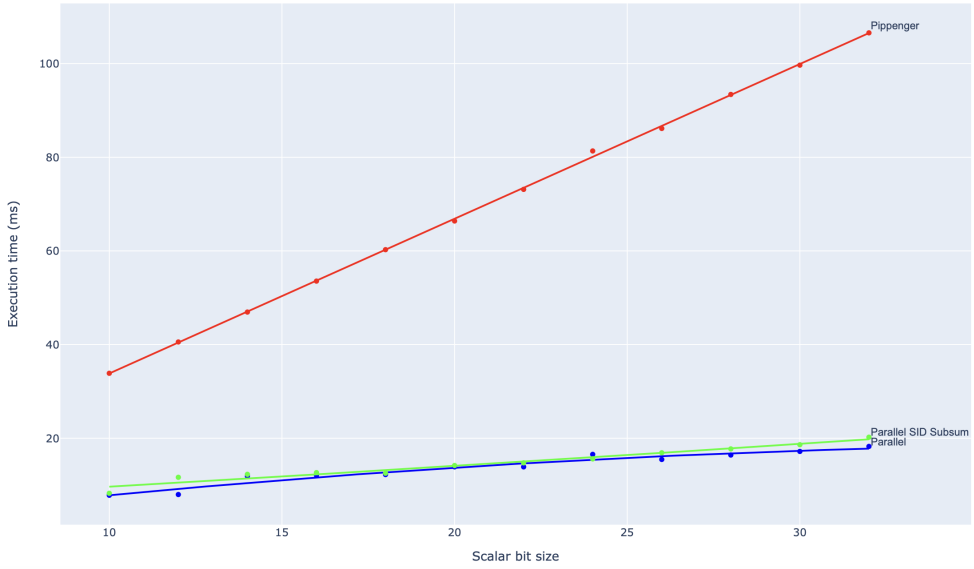

Figure 6.1: Best performing algorithms across scalar sizes ($n = 1000$, $c = 2$)

Scalar sizes: Figure 6.1 depicts the execution times of the three algorithms as a function of scalar size for $c = 2$ and $n = 1000$. Barring the nuances of parallelism due to hardware configurations, all these three algorithms demonstrate near linear relationships with scalar sizes although differences in slope are significant. Impressively, parallelism alone demonstrates the best performance (marginally) across the board, with $7.817ms$ for $b = 10$ only increasing to $18.246ms$ for $b = 32$. We note that this outperformance is due to the small window size which does not show the true potential of both the SID and novel subsum accumulation optimisations. Parallel SID Subsum also displays notable efficiency with $8.235ms$ for $b = 10$ only increasing to $20.215ms$ for $b = 32$, showing a steady increase in % improvement on the Pippenger method from 76% to 81%.

Window size: Figure 6.2 presents the execution times across different window sizes, keeping $n = 1000$ and $b = 32$ constant. The runtime of Pippenger's method initially

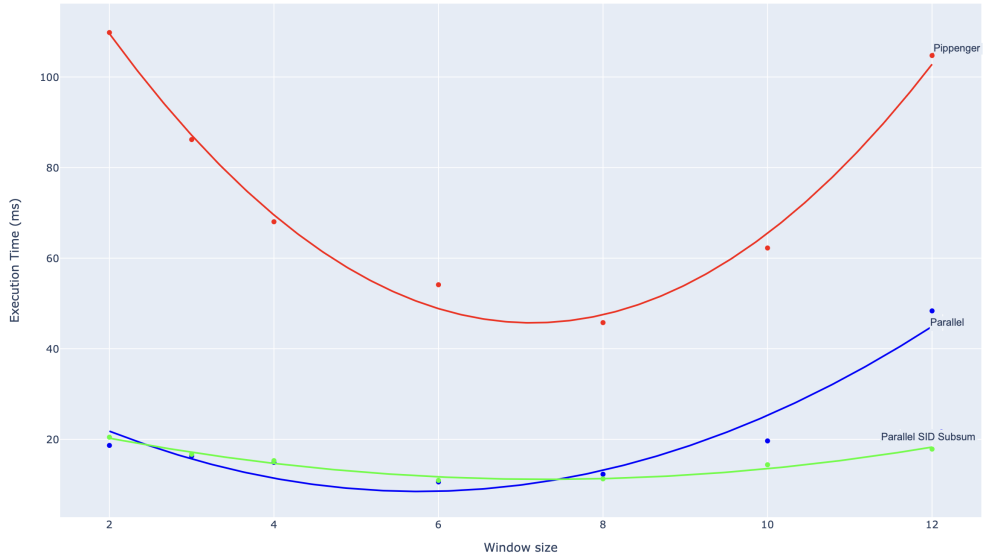

Figure 6.2: Best performing algorithms across window sizes ($n = 1000$, $b = 32$)

decreases with window size but ultimately shows a pronounced U-shaped trend due to unnecessary additions in the subsum accumulation phase as $c \geq 8$. Parallelism's impact here is limited by number of cores but it is Parallel SID Subsum that demonstrates remarkable efficiency with increasing window size. While its performance slightly lags behind Parallel for $c \leq 6$ (when parallelism achieves its peak performance), it exhibits an impressive improvement for $c \geq 8$, as shown by the table below. These observations can be attributed to the correlation between empty buckets and optimisation of subsum accumulation.

Algorithms	**2**	**3**	**4**	**6**	**8**	**10**	**12**	**16**
Parallel	18.65	16.32	14.90	10.59	12.29	19.65	48.36	606.34
Parallel SID Subsum	20.47	16.69	15.29	10.94	11.27	14.38	17.84	21.40
Improvement (%)	−9.74	−2.23	−2.59	−3.25	8.28	26.80	63.11	96.47

Table 6.3: Parallel SID Subsum vs. Parallel for $2 \leq c \leq 16$ ($b = 32$ and $n = 1000$)

Number of point-scalar pairs: The effect of number of point-scalar pairs on runtimes has one key trend: As stated in Section 6.1, all algorithms produce runtimes with an approximately one-to-one relationship with n increases by for $n \geq 10^4$. We believe this can be attributed to the increase in complexity of the MSM problem with n. Notably, for our 2 best performing algorithms, the difference is attributed to the number of empty buckets. For $n \leq 10^4$, Parallel SID Subsum produce the best performance. However as n exceeds 10^4, the likelihood of a large number of empty buckets for $n \geq 10^4$ is smaller and we see

outperformance from parallelism alone. These variances are presented in Table 6.4.

Number of points	**10**	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**	**10^7**
Parallel	4.10	3.62	12.45	97.94	986.35	9621.01	95476.00
Parallel SID Subsum	1.02	2.00	11.25	100.33	991.28	9913.84	99048.86
Improvement (%)	75.01	44.83	9.63	-2.45	-0.50	-0.30	-0.37

Table 6.4: Parallel SID Subsum vs. Parallel for $10 \leq n \leq 10^7$ ($b = 32$ and $c = 8$)

Comparison with BLS Curves: Precise comparison between MSM algorithm performance on BLS and MNT curves, while desired, is challenging due to the lack of data available that offers a like-for-like comparison in terms of algorithm, hardware and parameters. Youssef El-Housni's PhD thesis in 2023 [25] does however present that the Parallel SID Pippenger computes an MSM with $n = 2^{16}$ $\mathbb{G}_1$ points in $509ms$ on a BLS12-377 curve. Our implementation of the same algorithm with solves an MSM with same n on MNT4-298 curves in $646ms$, which suggests little sacrifice in efficiency. There are, however, certain factors that complicate direct comparisons. The experiments in El-Housni's thesis were tested on a Samsung Galaxy A13 5G smartphone and the implementation used the Twisted Edwards curve form ($a = -1$) and an extended coordinate system.

Overall, we conclude that there is no one-fit-all algorithm that is best for any configuration and the quest for the most optimal algorithm-optimisation combination depends and all three parameters: window size (c), bit size of scalar (b), and the number of point-scalar pairs (n). Without any optimisations or precomputations, the Pippenger bucket method far outperforms alternatives, especially as the size of the problem gets large, crucial in zk-SNARK pairing-based trusted setups.

In terms of optimisations to it, the potential of parallelism is only limited by the number of performance cores and should be a mainstay regardless of parameter combinations or other optimisations. On the other hand, SID and the novel subsum accumulation [38] provide further optimisation in certain conditions.

For SID, c being large enough makes the effect from reduction of buckets more pronounced. Our results show that integrating SID when $c \geq 8$ results in improvements in 87.5% of 14 cases. The two cases, are when the novel subsum accumulation is present and parallelism is not as these two optimisations counter-act each other and this counter-acting is not mitigated by parallelism.

Finally, for the novel subsum accumulation algorithm, the effects depend on the interaction between c and n. With c large enough and n small enough, we have a high expected number of empty buckets. In these conditions, its improvement on the naïve subsum accumulation in the Pippenger bucket method are noticeable. The uniqueness of this optimisation is twofold: First, it makes up for a weakness in the bottleneck step within the Pippenger bucket method, something that our other 2 optimisations cannot offer and second, even in unfavourable conditions (small c, large n), it introduces marginal additional cost, signifying the efficiency and robustness of this optimisation.

Chapter 7

Conclusion and future work

In this report, we embarked on a comprehensive exploration and investigation into the efficiency of various MSM algorithms, with a particular focus on their performance on MNT curves. Beginning with foundational MSM algorithms like the double-and-add and the Montgomery multiplication trick, we delved into more sophisticated methods such as the Bos-Coster, Straus, and Pippenger bucket method. Our analysis included an in-depth survey and theoretical exploration of these algorithms, offering novel interpretations with hopes of serving as a reference point for future cryptographic research. We also investigated several optimisations to the Pippenger bucket method including parallelism, signed integer decomposition, and a novel subsum accumulation algorithm [38].

The implementation of key algorithms and optimisations was carried out in Rust on the MNT4-298 curve within the Arkworks framework. Through thorough experiments and detailed evaluations, we provided insights into the efficiency and scalability of these algorithms across three critical parameters, pinpointing optimal configuration of these parameters for each algorithm. With our detailed experiment results, this report establishes a benchmark for evaluating MSM algorithms on MNT curves, addressing a notable gap in current literature.

However, our investigation also encountered certain limitations. Firstly, our investigation does not extend to MSM algorithms involving precomputation. We omitted this due to storage limitations and hardware constraints that also prevented a thorough exploration of hardware-based optimisations and their potential impacts. Additionally, our algorithms were implemented on the MNT4-298 curve. It is crucial to note that while it served our

purpose of understanding the nuances of efficiency of MSM algorithms, this curve offers a lower security level and is not suitable for deployment in cryptographic applications. Finally, the scope of our analysis was bounded by our hardware's capabilities, restricting the range of the number of point-scalar pairs that we considered.

Future work could focus on multiple areas. Firstly, future studies could explore how pre-computation strategies and hardware-based optimisations could enhance the performance of MSM algorithms on MNT curves as well as how these algorithms scale for a greater number of point-scalar pairs and larger scalar sizes.

Moreover, since the improvement given by the novel subsum accumulation optimisation [38] is positively correlated to the number of empty buckets, we believe that studying more precisely the relationship between the expected value of empty buckets and runtime could provide valuable insights into this optimisation. By employing this problem as a classic balls into bins problem, where the balls correspond to the points and the bins to the buckets, we get that this expected value equals to $b \cdot (1 - \frac{1}{b})^n$ where b is the number of buckets, n the number of scalar-point pairs, $(1 - \frac{1}{b})^n$ is the probability a specific bucket is empty after distributing n points.

Finally, we believe that looking at solving the c-bit MSMs after partitioning in the Pippenger bucket method as a linear algebra problem deserves further consideration. What we mean by this is once we have the $\lceil \frac{b}{c} \rceil$ c-bit MSMs, we could construct a matrix M of size ($\lceil \frac{b}{c} \rceil$ x n), where each row corresponds to a segment of the n scalars and each column corresponds to a scalar's $\lceil \frac{b}{c} \rceil$ segments and a column vector v which contains n points. The c-bit MSM computation (step 2 in the Pippenger bucket method) can then be modeled as a matrix multiplication where the resulting point would equal the scaling and doubling of points in $M \cdot v$. However, we did not explore this further.

Bibliography

[1] R. Gennaro *et al.*, "Quadratic span programs and succinct nizks without pcps," in *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, Springer, 2013, pp. 626–645.

[2] *Ethereum: A community-run technology powering the cryptocurrency ether (eth) and thousands of decentralized applications.* https://ethereum.org/en/, Accessed: 4 April 2024.

[3] *Zcash: A simple, secure digital currency that protects your privacy.* https://z.cash/, Accessed: 4 April 2024.

[4] A. Miyaji *et al.*, "New explicit conditions of elliptic curve traces for fr-reduction," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E84-A, no. 5, pp. 1234–1243, 2001.

[5] R. Shen *et al.*, "Accelerating zk-snark with group and zone optimization on gpu," in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2023, pp. 24–31.

[6] D. Johnson *et al.*, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, pp. 36–63, 2001.

[7] D. J. Bernstein *et al.*, *Faster batch forgery identification*, Cryptology ePrint Archive, Paper 2012/549, https://eprint.iacr.org/2012/549, 2012. [Online]. Available: https://eprint.iacr.org/2012/549.

[8] W. Diffie *et al.*, "New directions in cryptography," in *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, 2022, pp. 365–390.

[9] *Zero-knowledge proofs: Mooc, spring 2023.* https://zk-learning.org/, Accessed: 30 October 2023.

[10] B. Stone, *Rust essential training*, https://www.linkedin.com/learning/rust-essential-training/learn-rust-programming?u=69919578, Accessed: 20 November 2023.

[11] *Arkworks: An ecosystem for developing and programming with zksnarks.* https://github.com/arkworks-rs, Accessed: 25 November 2023.

[12] V. Miller, "Short programs for functions on curves," Unpublished manuscript, 1986.

[13] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, issue. 177, pp. 203–209, Jan. 1987, http://dx.doi.org/10.1090/S0025-5718-1987-0866109-5.

[14] D. Boneh *et al.*, "A graduate course in applied cryptography," *Draft 0.5*, 2020.

[15] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*, Springer, 2006, pp. 207–228.

[16] H. Cohen *et al.*, "Efficient elliptic curve exponentiation using mixed coordinates," in *Advances in Cryptology—ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings*, Springer, 1998, pp. 51–65.

[17] D. Mahto *et al.*, "Rsa and ecc: A comparative analysis," *International Journal of Applied Engineering Research*, vol. 12, no. 19, pp. 9053–9061, 2017.

[18] M. Scott, "Implementing cryptographic pairings," *Lecture Notes in Computer Science*, vol. 4575, p. 177, 2007.

[19] D. Boneh *et al.*, "Identity-based encryption from the weil pairing," in *Annual International Cryptology Conference*, Springer, 2001, pp. 213–229.

[20] P. S. L. M. Barreto *et al.*, "Pairing-friendly elliptic curves of prime order," in *International workshop on selected areas in cryptography*, Springer, 2005, pp. 319–331.

[21] P. S. L. M. Barreto *et al.*, "Efficient algorithms for pairing-based cryptosystems," in *Advances in Cryptology 2002*, https://doi.org/10.1007/3-540-45708-9_23, 2002.

[22] R. Blum *et al.*, "Superlight – a permissionless, light-client only blockchain with self-contained proofs and bls signatures," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 36–41.

[23] P. Wang *et al.*, "Environmental adaptive privacy preserving contact tracing system: A construction from public key rerandomizable bls signatures," *IEEE Access*, vol. 10, pp. 37 181–37 199, 2022, https://doi.org/10.1109/ACCESS.2022.3164186.

[24] K. Karabina *et al.*, "On prime-order elliptic curves with embedding degrees k= 3, 4, and 6," in *Algorithmic Number Theory: 8th International Symposium, ANTS-VIII Banff, Canada, May 17-22, 2008 Proceedings 8*, Springer, 2008, pp. 102–117.

[25] Y. E. Housni, "The arithmetic of pairing-based proof systems," Ph.D. dissertation, Institut Polytechnique de Paris, May 2023.

[26] A. Chiesa *et al.*, "On cycles of pairing-friendly elliptic curves," *SIAM Journal on Applied Algebra and Geometry*, vol. 3, no. 2, pp. 175–192, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1803.02067.

[27] X. Liu *et al.*, "Scalable collaborative zk-snark: Fully distributed proof generation and malicious security," *Cryptology ePrint Archive*, 2024.

[28] D. Hankerson *et al.*, "Elliptic curve cryptography," in *Encyclopedia of Cryptography, Security and Privacy*, Springer, 2021, pp. 1–2.

[29] R. R. Ahirwal *et al.*, "Elliptic curve diffie-hellman key exchange algorithm for securing hypertext information on wide area network," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 2, pp. 363–368, 2013.

[30] E. Ben-Sasson *et al.*, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Annual Cryptology Conference*, Springer, 2013, pp. 90–108.

[31] I. Meckler *et al.*, "Coda: Decentralized cryptocurrency at scale," *O (1) Labs Whitepaper*, vol. 10, p. 4, 2018.

[32] Y. E. Housni *et al.*, "Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition," in *International Conference on Cryptology and Network Security*, Springer, 2020, pp. 259–279.

[33] S. Bowe *et al.*, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 947–964.

[34] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, IEEE Computer Society, 1976, pp. 258–263.

[35] J. W. Bos *et al.*, "Post-quantum key exchange for the tls protocol from the ring learning with errors problem," in *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 553–570.

[36] E. G. Straus, "Addition chains of vectors (problem 5125)," *American Mathematical Monthly*, vol. 70, pp. 806–808, 1964.

[37] Y. E. Housni *et al.*, "Edmsm: Multi-scalar-multiplication for snarks and faster montgomery multiplication," *Cryptology ePrint Archive*, 2022, https://eprint.iacr.org/2022/1400.

[38] G. Luo *et al.*, "Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 2, pp. 358–380, 2023, https://doi.org/10.46586/tches.v2023.i2.358-380.

[39] A. Ray *et al.*, "Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024, pp. 33–39.

[40] K. Aasaraai *et al.*, "Fpga acceleration of multi-scalar multiplication: Cyclonemsm," *Cryptology ePrint Archive*, 2022.

[41] X. Zhu *et al.*, "Elastic msm: A fast, elastic and modular preprocessing technique for multi-scalar multiplication algorithm on gpus," *Cryptology ePrint Archive*, 2024.

[42] P. D. Rooij, "Efficient exponentiation using precomputation and vector addition chains," in *Workshop on the Theory and Application of Cryptographic Techniques*, Springer, 1994, pp. 389–399.

[43] J. L. Hennessy *et al.*, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[44] D. Culler *et al.*, *Parallel Computer Architecture: A Hardware/Software Approach*. Gulf Professional Publishing, 1999.

[45] D. F. Aranha *et al.*, "A survey of elliptic curves for proof systems," *Designs, Codes and Cryptography*, vol. 91, no. 11, pp. 3333–3378, 2023.

[46] H. Liu, *Elliptic curves and integer factorization*, https://www.math.uchicago.edu/~may/VIGRE/VIGRE2011/REUPapers/LiuH, Retrieved online on 3rd March 2024, 2011.

[47] I. Setiadi *et al.*, "Elliptic curve cryptography: Algorithms and implementation analysis over coordinate systems," in *2015 2nd International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, https://doi.org/10.1109/ICAICTA.2015.7335349, 2015, pp. 1–6.

[48] D. Naccache *et al.*, "Projective coordinates leak," in *Advances in Cryptology – EUROCRYPT 2004*, https://doi.org/10.1007/b97182, 2004.

[49] M. Joye *et al.*, "Protections against differential analysis for elliptic curve cryptography—an algebraic approach—," in *Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop 2001*, https://doi.org/10.1007/3-540-44709-1_31, 2001, pp. 377–390.

[50] D. J. Bernstein *et al.*, "Faster addition and doubling on elliptic curves," in *Advances in Cryptology (ASIACRYPT) 2007: 13th International Conference on the Theory and Application of Cryptology and Information Security*, https://doi.org/10.1007/978-3-540-76900-2_3, 2007, pp. 29–50.

[51] T. Izu *et al.*, "Exceptional procedure attack on elliptic curve cryptosystems," in *Public Key Cryptography—PKC 2003: 6th International Workshop on Practice and Theory in Public Key Cryptography 2003*, https://doi.org/10.1007/3-540-36288-6_17, 2002, pp. 224–239.

[52] G. Barthe *et al.*, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 328–343, 2018, https://doi.org/10.1109/CSF.2018.00031.

[53] D. J. Bernstein *et al.*, "Twisted edwards curves," *Progress in Cryptology - AFRICACRYPT 2008: First International Conference on Cryptology in Africa*, pp. 389–405, Jun. 2008, http://doi.org/10.1007/978-3-540-68164-9_26.

[54] H. Hisil *et al.*, *Jacobi quartic curves revisited*, Cryptology ePrint Archive, https://eprint.iacr.org/2009/312, 2009.

Appendix A

Supplementary material on elliptic curve mathematics

A.1 Chord-and-tangent rule

Given an elliptic curve E over a field $\mathbb{F}_p$ and two points $P, Q \in E(\mathbb{F}_p)$, point addition $\oplus$ is the addition of P and Q to yield a third point $P \oplus Q = R \in E(\mathbb{F}_p)$. Point doubling is a special case of point addition where $P = Q$ which yields a point $P + P = S \in E(\mathbb{F}_p)$. Point doubling can involve slightly fewer field operations than general point addition by leveraging algebraic properties but for simplicity of considering complexities, it is common in ECC literature to assume their costs are equal. Scalar multiplication is the multiplication of a point P by a scalar k which yields a point $k \cdot P = R \in E(\mathbb{F}_p)$. It is the most computationally intensive operation in ECC and is effectively either a series of point doublings and additions, determined by the binary representation of the scalar.

The operations on an elliptic curve can be geometrically visualised using the "chord-and-tangent" rule, which is depicted in Figure A.1.

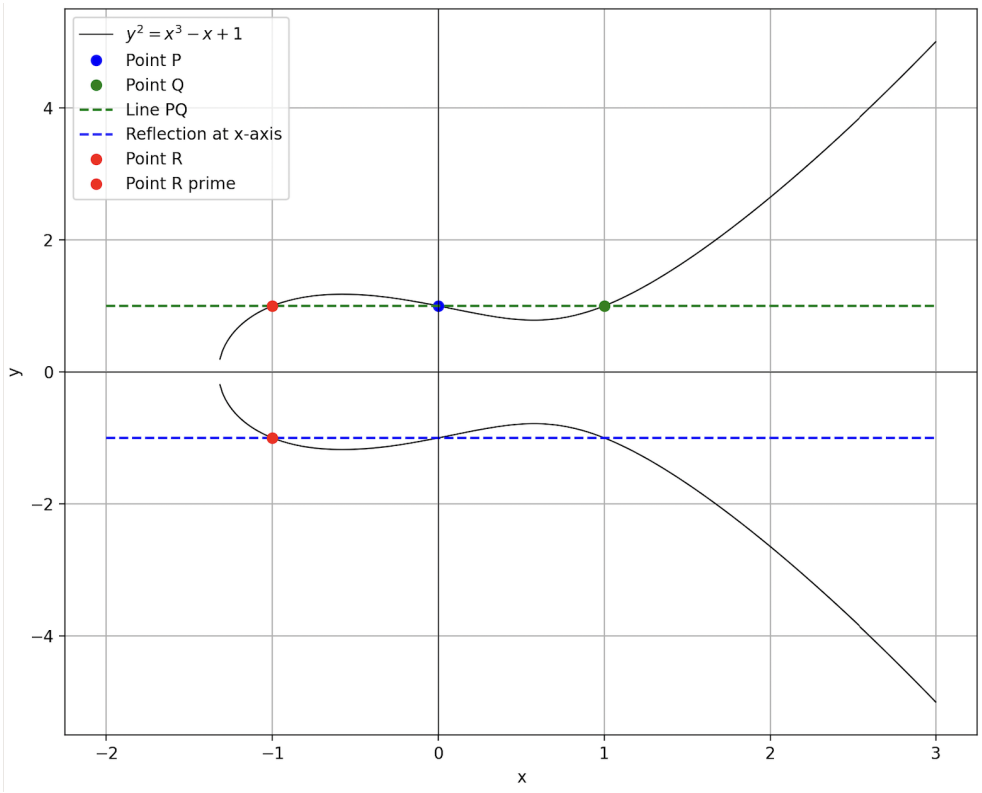

Figure A.1: Visualisation of the chord-and-tangent rule on an elliptic curve.

The chord-and-tangent rule operates as follows: given two points P and Q on an elliptic curve, if $P \neq Q$, the line intersecting P and Q will intersect the curve at exactly one additional point R. By reflecting R across the x-axis to obtain R', we define the sum of P and Q as $P + Q = R'$. If $P = Q$, we have the point doubling operation where we consider the line tangent to P intersects the curve at another point, and reflecting this point across the x-axis gives us the result of point doubling, $2P$.

Mathematically, these operations are governed by explicit formulae derived from the Weierstrass equation. The formulae ensure that if P and Q are points on the curve, the result R will also lie on the curve, maintaining the group closure property. A deeper explanation of the chord-tangent composition can be found in [46], [47].

The algebraic structure that follows from this geometric rule ensures the curve can be used for secure cryptographic applications, which rely heavily on the computational difficulty of reversing the point addition process, which brings us to our next point.

A.2 Coordinate systems

Elliptic curves can be represented in various coordinate systems. This choice has a direct impact on the performance of cryptographic algorithms. Therefore, various coordinate systems have been devised to optimise operations of point addition and doubling. A comparison of the two broad classes of coordinate systems from a conference proceeding by Iskandar Setiadi [47] is presented in the definitions below.

Definition A.1 (Affine coordinates [47])**.** *The canonical coordinate system, the affine coordinate system expresses a point P on an elliptic curve as a pair (x, y) satisfying the curve's equation.*

While conceptually simple, affine coordinates require inverse operations in the field, which are computationally expensive. To mitigate this, projective coordinate systems were introduced, eliminating the need for inversion operations (which correspond to the last step of the geometrical operation depicted by Figure A.1), replacing them with multiplications, which are significantly faster in practice.

Definition A.2 (Projective coordinates [47])**.** *A projective coordinate system expresses a point P as a triplet $(X, Y, Z) = (\lambda x, \lambda y, \lambda)$, $\lambda \neq 0$, related to affine coordinates by $x = X/Z$ and $y = Y/Z$.*

Building upon the efficiency of projective coordinates, Jacobian coordinates offer a further optimisation specifically advantageous for curves defined in the Weierstrass form.

Definition A.3 (Jacobian coordinates [48])**.** *A Jacobian coordinate system expresses a point P on an elliptic curve as a triplet $(X, Y, Z) = (\lambda^2 x, \lambda^3 y, \lambda)$, $\lambda \neq 0$, related to affine coordinates by $x = X/Z^2$ and $y = Y/Z^3$.*

Furthermore, mixed coordinate systems which use different coordinate systems for different points during the elliptic curve arithmetic operations, have shown to significantly reduce the computational overhead of the operations while increasing security. [49]–[51]

Finally, an extended coordinate system is one that introduces additional auxilary variables to further optimise the number of required operations. These systems take many forms and can sometimes allow for unified addition formulas that cover all cases, including point doubling. Common examples are the extended Edwards and Jacobian coordinate

systems, which are tailored to curves written in these forms. We will study these equations of elliptic curves in more detail in the next subsection.

A.3 The various elliptic curves forms

The equations of elliptic curves can be written in various forms. We have looked at one such form, the short Weierstrass form. The choice of an equation is a nuanced decision that depends on the specific cryptographic application at hand. Below, we describe some common equations used in practice.

Introduced by Harold Edwards and applied to cryptography by D. J. Bernstein and T. Lange [50], the Edwards curve offers a symmetric form defined as

$$x^2 + y^2 = c^2(1 + dx^2y^2), \tag{A.1}$$

where c and d are elements in the field $\mathbb{F}_p$ with c^2 being a non-zero square and d a non-square, ensuring the curve is non-singular. The Edwards curve is notable for its simple and symmetric formula, which supports a unified addition law for all pairs of points. A unified addition law means that the formulas for point addition and doubling are the same for all points on the curve, including the identity and points of order two. There are no exceptions or special cases to handle, unlike in other forms such as the Weierstrass form, where special cases can complicate implementations. This complete group law enables more efficient scalar multiplication as algorithms wouldn't need to include conditional branches to handle special cases in point addition or doubling, enabling constant-time implementations, crucial to resist timing-related side-channel attacks [52].

Building on the structure of Edwards curves, the twisted Edwards form, as introduced by Bernstein et al. [53], is described by the equation

$$ax^2 + y^2 = 1 + dx^2y^2, \tag{A.2}$$

where parameters a and d in $\mathbb{F}_p$ are chosen such that a is a square and d is a non-square. This generalisation of the original Edwards curve allows for the representation of a broader class of elliptic curves while maintaining the advantage of a complete group law.

Another popular equation is the Jacobi quartics form which provides an alternative representation with an added coordinate to optimise elliptic curve arithmetic. They are defined by the equation

$$y^2 = dx^4 + 2ax^2 + 1, \tag{A.3}$$

where a and d are non-zero elements of $\mathbb{F}_p$ chosen to avoid singularity. This form doesn't support a unified addition law but allows for more efficient computation of the doubling and addition of points [54].

These and various other elliptic curve forms and its subtle differences embody the confluence of algebraic structure and cryptographic utility. They each present trade-offs in terms of efficiency, implementation complexity, and security properties.

Appendix B

Complete experiment results

This appendix provides complete experiment results. We provide these results in two sections, one section is dedicated to the mean runtimes for each configuration and the other to the experiment results run-by-run as well as the statistical measures derived from these runs for each configuration. A couple of quick notes:

1. All runtimes for all sections are rounded up to 3 decimal places and are presented in milliseconds (ms).
2. Naïve MSM here refers to the process of direct scalar multiplication, not the naïve approach of repeated point additions discussed in Section 4.2.
3. The optimisations are all implemented on top of the Pippenger bucket method and in the table of results, are referred to as Parallel, SID and Subsum respectively. Parallel refers to a parallelised Pippenger, SID to Pippenger with signed integer decomposition and Subsum to Pippenger with the novel subsum accumulation optimisation [38]. Therefore, Parallel SID Subsum would refer to the Pippenger bucket method with all three optimisations and so on.
4. Mean μ is calculated as $\frac{1}{n}\sum_{i=1}^{n} x_i$.
5. Standard error SE is calculated as $\frac{\sigma}{\sqrt{n}}$).
6. Modified coefficient of variation CV is calculated as $(\frac{\sum_{i=1}^{n}|x_i-\mu|}{n\cdot\mu} \times 100$.
7. Normalised peak deviation NPD is calculated as $\frac{\max(x_i)-\mu}{\mu}$)
8. For further information on statistical measures, readers may refer to Section 5.3.

B.1 Mean runtimes

Scalar sizes ($c = 2$ and $n = 1000$)

Algorithms	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Naïve	95.243	115.639	135.984	154.741	174.747	195.368
Trivial	95.004	115.841	134.606	154.076	173.854	193.713
Pippenger	33.865	40.551	46.940	53.552	60.265	66.389
Parallel	7.817	7.969	11.886	12.085	12.234	12.445
SID	40.293	46.999	53.322	59.837	67.002	72.906
Subsum	33.490	40.347	46.326	53.239	59.746	65.620
Parallel SID	8.332	11.876	12.432	12.686	12.967	14.360
Parallel Subsum	7.662	8.352	11.616	12.162	12.177	12.031
SID Subsum	39.869	46.911	52.785	59.684	66.187	72.611
Parallel SID Subsum	8.235	11.650	12.285	12.645	12.522	14.168

Algorithms	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}	2^{32}
Naïve	214.763	234.415	254.408	273.583	297.256	313.898
Trivial	216.946	233.403	252.962	277.021	292.167	312.955
Pippenger	73.140	81.346	86.152	93.416	99.672	106.571
Parallel	13.876	16.547	15.462	16.408	17.174	18.246
SID	79.946	86.431	92.783	100.009	109.434	112.727
Subsum	72.693	79.134	85.328	92.362	98.578	105.626
Parallel SID	14.685	15.899	16.883	17.668	18.688	20.139
Parallel Subsum	14.037	14.468	15.593	15.950	17.498	18.791
SID Subsum	79.510	86.549	92.345	99.425	105.426	112.575
Parallel SID Subsum	14.734	15.576	16.883	17.712	18.609	20.215

Number of point-scalar pairs ($c = 8$ and $b = 32$)

Algorithms	10	10^2	103	10^4
Naïve	6.399	32.221	322.573	3214.348
Trivial	5.634	31.786	321.602	3206.444
Pippenger	13.781	12.837	46.461	372.466
Parallel	4.097	3.621	12.447	97.935
SID	6.541	8.880	46.064	421.378
Subsum	4.310	7.255	46.213	381.397
Parallel SID	2.126	2.591	11.373	100.242
Parallel Subsum	1.440	2.153	12.297	100.223
SID Subsum	2.778	7.047	46.289	425.327
Parallel SID Subsum	1.024	1.998	11.250	100.330

Algorithms	10^5	10^6	10^7
Naïve	31832.775	325361.028	3210780.467
Trivial	31871.805	322932.705	3166241.236
Pippenger	3680.357	37006.982	370484.350
Parallel	986.354	9621.009	95476.001
SID	4172.601	42075.658	421116.627
Subsum	3722.504	37633.895	376022.895
Parallel SID	984.946	9801.673	97381.459
Parallel Subsum	977.916	9776.131	97234.955
SID Subsum	4214.307	42547.409	424694.197
Parallel SID Subsum	991.280	9913.841	99048.864

Window sizes ($b = 32$ and $n = 1000$)

Algorithms	2	3	4	6
Naïve	319.392	319.443	316.188	318.071
Trivial	321.581	318.592	314.868	317.616
Pippenger	109.812	86.181	68.034	54.129
Parallel	18.650	16.322	14.901	10.590
SID	115.835	89.575	72.776	53.917
Subsum	106.673	86.146	68.059	54.299
Parallel SID	20.075	16.643	15.276	11.031
Parallel Subsum	18.581	16.510	14.968	11.137
SID Subsum	114.281	88.725	72.602	54.469
Parallel SID Subsum	20.467	16.686	15.288	10.935

Algorithms	8	10	12	16
Naïve	318.263	317.316	317.229	315.868
Trivial	317.940	314.690	316.409	315.608
Pippenger	45.762	62.240	104.740	1171.769
Parallel	12.289	19.646	48.355	606.339
SID	45.708	49.450	67.608	605.703
Subsum	45.560	51.824	47.109	45.029
Parallel SID	11.322	15.391	30.892	338.338
Parallel Subsum	12.350	16.237	18.695	23.808
SID Subsum	46.090	47.718	45.224	45.182
Parallel SID Subsum	11.271	14.381	17.838	21.396

B.2 Experiment results run-by-run

B.2.1 Bottleneck examination

Pippenger bucket method - segmented by phases ($c = 8, b = 32$ and $n = 1000$)

Step	1	2	3	4	5	6	7
Partitioning	0.369	0.323	0.343	0.309	0.315	0.333	0.361
Bucketing	1.573	1.538	1.436	1.478	1.423	1.459	1.639
Subsum	45.589	45.643	45.563	45.973	46.414	45.207	46.984
Combining	0.182	0.146	0.115	0.147	0.124	0.156	0.145
Total	47.712	47.650	47.457	47.907	48.276	47.155	49.129

Step	8	9	10	μ	% of runtime
Partition	0.372	0.320	0.341	0.339	0.708
Bucketing	1.482	1.534	1.595	1.516	3.168
Subsum	45.350	45.250	46.505	45.848	95.827
Combining	0.121	0.147	0.140	0.142	0.297
Total	47.325	47.251	48.581	47.844	100.000

Cost of signed integer decomposition ($c = 8, b = 32$ and $n = 1000$)

Step	1	2	3	4	5	6
SID Cost	0.127	0.126	0.136	0.125	0.128	0.134

Step	7	8	9	10	μ	SE
SID Cost	0.137	0.123	0.125	0.121	0.128	0.002

B.2.2 Scalar sizes

Naïve MSM

2^b	1	2	3	4	5	6	7
2^{10}	95.653	93.849	96.315	95.586	95.399	97.367	93.943
2^{12}	114.754	116.034	114.554	116.659	115.991	114.734	119.967
2^{14}	135.001	135.117	135.061	135.888	136.734	137.613	135.271
2^{16}	154.947	155.143	154.842	155.774	154.851	152.194	154.437
2^{18}	173.671	173.206	174.065	175.110	173.849	174.458	174.632
2^{20}	194.911	193.916	193.413	197.086	195.728	193.188	197.897
2^{22}	214.089	214.411	212.827	213.870	214.033	213.663	218.258
2^{24}	237.061	232.832	233.885	233.577	234.162	235.561	232.612
2^{26}	252.824	252.925	254.603	259.110	253.260	253.395	251.953
2^{28}	274.051	273.514	273.682	273.364	271.942	273.444	273.704
2^{30}	292.307	293.871	292.412	294.071	293.919	293.490	299.905
2^{32}	314.902	313.108	313.270	314.689	312.496	312.811	312.338

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	94.478	95.465	94.376	95.243	0.909	0.350	0.022
2^{12}	114.461	114.478	114.764	115.639	1.054	0.541	0.037
2^{14}	139.334	134.506	135.319	135.984	0.842	0.473	0.025
2^{16}	154.284	156.287	154.652	154.741	0.439	0.341	0.010
2^{18}	173.976	181.123	173.385	174.747	0.771	0.731	0.036
2^{20}	193.552	200.925	193.063	195.368	1.041	0.816	0.028
2^{22}	213.373	219.470	213.635	214.763	0.764	0.703	0.022
2^{24}	232.678	239.282	232.502	234.415	0.739	0.711	0.021
2^{26}	254.142	253.605	258.265	254.408	0.688	0.751	0.018
2^{28}	271.794	274.067	276.272	273.583	0.282	0.390	0.010
2^{30}	325.164	295.494	291.923	297.256	2.056	3.184	0.094
2^{32}	313.220	318.072	314.076	313.898	0.392	0.539	0.013

Trivial MSM

2^b	1	2	3	4	5	6	7
2^{10}	93.880	94.062	97.199	96.263	94.769	96.233	94.886
2^{12}	113.436	116.600	117.928	115.161	115.109	120.990	114.457
2^{14}	133.790	133.802	133.442	136.583	135.940	135.703	135.276
2^{16}	156.347	153.714	153.860	154.418	152.323	151.164	153.080
2^{18}	172.405	172.016	173.022	174.030	172.090	173.301	176.142
2^{20}	193.975	193.228	192.366	196.463	192.989	192.498	197.451
2^{22}	212.469	212.976	211.538	213.148	214.358	212.474	215.151
2^{24}	234.676	231.859	232.753	233.149	232.877	233.018	231.281
2^{26}	252.139	251.505	253.416	253.994	252.237	251.723	251.524
2^{28}	272.134	272.560	272.007	285.751	270.960	271.761	272.619
2^{30}	291.593	292.409	291.006	292.580	291.953	291.816	292.060
2^{32}	313.196	311.657	311.757	313.508	310.963	310.884	311.633

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	93.504	94.743	94.499	95.004	0.986	0.376	0.023
2^{12}	114.211	117.357	113.164	115.841	1.642	0.762	0.044
2^{14}	134.174	133.527	133.823	134.606	0.754	0.365	0.015
2^{16}	153.053	159.273	153.532	154.076	1.014	0.718	0.034
2^{18}	172.333	179.481	173.719	173.854	0.931	0.737	0.032
2^{20}	192.781	193.868	191.509	193.713	0.713	0.591	0.019
2^{22}	212.495	253.073	211.781	216.946	3.330	4.029	0.167
2^{24}	234.123	239.056	231.237	233.403	0.655	0.719	0.024
2^{26}	253.506	255.087	254.493	252.962	0.449	0.413	0.008
2^{28}	302.318	272.319	277.781	277.021	2.512	3.147	0.091
2^{30}	290.902	297.010	290.337	292.167	0.376	0.582	0.017
2^{32}	314.241	316.487	315.227	312.955	0.504	0.603	0.011

Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	33.699	34.603	34.584	33.854	33.613	34.075	33.196
2^{12}	40.021	40.759	42.185	40.441	40.243	40.817	39.346
2^{14}	46.469	46.561	46.723	47.544	47.444	47.201	46.561
2^{16}	54.139	53.208	53.722	53.625	53.141	51.909	52.572
2^{18}	59.281	59.343	59.935	60.647	60.006	59.784	63.575
2^{20}	66.848	66.324	65.753	66.421	66.424	65.857	65.982
2^{22}	72.615	72.998	72.114	73.353	73.538	72.272	73.719
2^{24}	80.467	79.235	94.056	79.335	79.863	79.982	79.018
2^{26}	86.430	86.305	86.580	87.065	86.079	86.289	85.676
2^{28}	93.555	92.967	92.573	94.214	92.302	92.368	94.288
2^{30}	98.998	99.389	99.029	99.999	99.870	99.537	99.256
2^{32}	106.794	105.989	106.104	106.668	107.837	105.070	105.715

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	33.197	34.155	33.670	33.865	1.157	0.157	0.022
2^{12}	40.365	41.812	39.521	40.551	1.661	0.285	0.040
2^{14}	46.966	47.258	46.670	46.940	0.731	0.126	0.013
2^{16}	53.278	56.343	53.581	53.552	1.364	0.368	0.052
2^{18}	59.464	60.827	59.792	60.265	1.411	0.402	0.055
2^{20}	66.751	67.569	65.957	66.389	0.624	0.176	0.018
2^{22}	72.814	75.354	72.622	73.140	0.931	0.297	0.030
2^{24}	79.278	83.699	78.524	81.346	3.704	1.484	0.156
2^{26}	86.116	85.663	85.317	86.152	0.443	0.160	0.011
2^{28}	93.427	93.587	94.875	93.416	0.739	0.275	0.016
2^{30}	99.139	102.613	98.894	99.672	0.695	0.347	0.030
2^{32}	106.561	107.429	107.542	106.571	0.641	0.277	0.012

Parallel Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	8.059	7.717	7.880	7.864	7.816	8.041	7.546
2^{12}	7.654	8.020	8.020	8.286	7.747	8.681	7.576
2^{14}	12.219	12.199	11.744	11.678	11.931	11.982	11.659
2^{16}	12.248	12.123	11.981	12.126	12.073	11.729	12.057
2^{18}	12.173	11.940	12.166	12.144	12.148	12.248	12.316
2^{20}	12.838	13.856	12.027	12.247	12.339	12.040	12.058
2^{22}	13.763	14.431	13.273	14.236	14.145	13.762	14.022
2^{24}	14.170	13.896	34.791	14.570	14.576	14.718	14.298
2^{26}	15.170	15.644	17.123	15.593	14.975	15.287	15.416
2^{28}	15.764	15.935	16.944	14.930	15.565	16.585	17.945
2^{30}	17.687	16.781	17.114	16.973	16.779	17.882	17.320
2^{32}	17.094	18.811	17.039	18.245	19.084	18.898	17.313

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	7.761	7.709	7.781	7.817	1.468	0.049	0.031
2^{12}	7.780	7.870	8.052	7.969	3.054	0.104	0.089
2^{14}	12.063	11.640	11.744	11.886	1.622	0.070	0.028
2^{16}	12.501	11.907	12.107	12.085	1.123	0.064	0.034
2^{18}	12.521	12.473	12.214	12.234	1.015	0.053	0.023
2^{20}	12.423	12.545	12.078	12.445	3.060	0.177	0.113
2^{22}	13.591	14.018	13.517	13.876	2.125	0.113	0.040
2^{24}	14.234	15.897	14.322	16.547	22.051	2.034	1.103
2^{26}	15.355	14.577	15.478	15.462	2.575	0.210	0.107
2^{28}	17.632	16.671	16.114	16.408	4.551	0.296	0.094
2^{30}	17.483	16.616	17.110	17.174	1.949	0.131	0.041
2^{32}	18.159	20.568	17.249	18.246	4.798	0.357	0.127

SID Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	40.081	40.906	40.391	39.747	40.356	41.209	40.126
2^{12}	46.162	47.003	48.012	47.339	46.932	48.212	46.518
2^{14}	52.590	53.487	53.227	53.898	53.877	53.489	53.333
2^{16}	61.014	59.249	59.537	60.206	59.760	59.552	59.242
2^{18}	66.405	66.312	66.327	67.098	66.275	66.930	70.243
2^{20}	72.908	73.105	72.199	72.294	73.125	72.675	72.692
2^{22}	78.995	79.742	79.530	79.247	79.886	79.117	80.995
2^{24}	86.149	86.566	86.888	86.447	85.637	86.647	86.388
2^{26}	93.452	92.673	93.147	93.162	92.608	93.079	92.301
2^{28}	99.409	99.430	99.733	98.952	99.127	100.262	100.598
2^{30}	105.151	106.096	106.541	107.111	106.330	106.033	105.059
2^{32}	112.328	112.427	113.304	113.417	113.795	112.072	111.400

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	39.877	40.487	39.752	40.293	0.934	0.153	0.023
2^{12}	47.095	46.237	46.481	46.999	1.135	0.221	0.026
2^{14}	53.082	53.457	52.782	53.322	0.603	0.134	0.011
2^{16}	60.287	60.212	59.315	59.837	0.792	0.183	0.020
2^{18}	65.820	68.388	66.222	67.002	1.410	0.425	0.048
2^{20}	73.086	74.044	72.930	72.906	0.484	0.163	0.016
2^{22}	80.499	81.153	80.300	79.946	0.791	0.242	0.015
2^{24}	85.557	87.948	86.081	86.431	0.542	0.215	0.018
2^{26}	92.321	92.868	92.217	92.783	0.387	0.134	0.007
2^{28}	99.788	100.736	102.058	100.009	0.723	0.295	0.020
2^{30}	139.520	107.156	105.341	109.434	5.499	3.351	0.275
2^{32}	112.539	112.465	113.521	112.727	0.555	0.239	0.009

Subsum Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	33.241	34.178	33.381	33.753	33.316	33.291	32.993
2^{12}	39.270	40.802	41.331	39.719	40.101	43.287	39.071
2^{14}	46.128	46.141	45.958	46.149	47.278	46.475	46.179
2^{16}	55.271	52.930	53.109	54.582	53.087	51.613	52.160
2^{18}	58.951	58.775	59.306	60.094	59.591	59.218	62.385
2^{20}	66.299	65.761	65.088	65.484	65.979	65.194	64.719
2^{22}	72.239	72.478	73.020	72.453	72.517	71.703	72.851
2^{24}	79.380	78.606	79.481	80.527	78.913	78.947	78.423
2^{26}	85.282	85.657	85.837	85.835	85.079	85.437	84.542
2^{28}	92.749	92.363	92.038	91.922	91.852	91.500	92.545
2^{30}	98.055	98.741	98.304	98.844	99.101	98.512	98.127
2^{32}	105.647	105.337	106.985	105.347	105.083	103.945	104.369

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	33.607	33.795	33.343	33.490	0.820	0.109	0.021
2^{12}	40.531	40.221	39.139	40.347	2.261	0.402	0.073
2^{14}	46.521	46.102	46.329	46.326	0.561	0.119	0.021
2^{16}	52.932	53.491	53.209	53.239	1.363	0.334	0.038
2^{18}	59.020	60.753	59.371	59.746	1.337	0.347	0.044
2^{20}	65.990	66.253	65.437	65.620	0.664	0.166	0.010
2^{22}	72.562	74.147	72.962	72.693	0.607	0.202	0.020
2^{24}	79.047	80.076	77.942	79.134	0.740	0.243	0.018
2^{26}	84.932	85.738	84.945	85.328	0.437	0.141	0.006
2^{28}	92.160	93.592	92.901	92.362	0.506	0.192	0.013
2^{30}	98.454	99.696	97.948	98.578	0.420	0.170	0.011
2^{32}	105.248	106.343	107.953	105.626	0.838	0.377	0.022

Parallel SID Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	8.357	8.778	8.268	8.754	8.175	8.160	8.216
2^{12}	11.771	12.189	11.862	11.632	11.963	11.963	11.678
2^{14}	12.501	12.179	13.181	12.152	12.686	12.365	12.100
2^{16}	12.609	12.460	12.797	12.707	12.616	13.217	12.461
2^{18}	12.552	12.903	12.941	13.031	12.452	12.759	12.854
2^{20}	14.599	14.722	13.696	14.214	14.597	13.996	14.987
2^{22}	14.805	14.175	14.505	14.695	14.569	14.542	14.666
2^{24}	15.354	16.840	17.100	15.442	14.460	15.293	17.383
2^{26}	15.704	18.638	16.093	15.329	17.948	17.792	15.823
2^{28}	18.306	18.180	17.121	16.673	18.272	17.690	17.573
2^{30}	18.472	17.714	19.636	18.084	19.164	18.824	17.681
2^{32}	20.524	19.617	19.912	20.673	19.528	20.345	20.230

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	8.138	8.052	8.418	8.332	2.354	0.080	0.054
2^{12}	11.436	12.387	11.882	11.876	1.688	0.087	0.043
2^{14}	12.557	12.158	12.436	12.432	1.935	0.104	0.060
2^{16}	12.973	12.647	12.377	12.686	1.495	0.081	0.042
2^{18}	12.666	13.184	14.333	12.967	2.537	0.167	0.105
2^{20}	13.879	14.298	14.615	14.360	2.393	0.131	0.044
2^{22}	15.902	14.397	14.592	14.685	1.836	0.146	0.083
2^{24}	16.141	15.635	15.342	15.899	4.866	0.297	0.093
2^{26}	17.136	16.132	18.233	16.883	6.317	0.381	0.104
2^{28}	17.643	17.068	18.156	17.668	2.561	0.180	0.036
2^{30}	19.357	19.068	18.882	18.688	2.999	0.215	0.051
2^{32}	19.925	20.617	20.020	20.139	1.682	0.128	0.027

Parallel Subsum Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	7.789	7.683	7.781	7.571	7.547	7.525	7.543
2^{12}	7.691	7.905	8.151	7.512	7.735	8.193	7.445
2^{14}	11.353	11.549	11.275	11.544	12.111	11.331	11.568
2^{16}	12.152	11.994	12.289	11.852	12.468	11.758	12.319
2^{18}	11.838	11.728	12.091	12.171	12.745	12.046	12.781
2^{20}	12.014	11.862	11.826	11.957	12.090	12.048	12.124
2^{22}	14.456	14.152	13.647	14.001	14.105	14.146	13.595
2^{24}	14.366	16.115	14.351	14.084	14.275	14.577	14.218
2^{26}	14.972	14.823	14.666	14.649	17.010	17.007	14.727
2^{28}	15.308	15.336	15.450	17.126	15.522	14.935	16.046
2^{30}	16.968	17.964	18.318	16.822	18.258	16.774	16.690
2^{32}	18.907	18.820	18.288	20.030	18.701	18.391	18.540

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	7.630	7.557	7.996	7.662	1.567	0.048	0.044
2^{12}	8.030	13.094	7.760	8.352	11.357	0.533	0.568
2^{14}	11.867	11.522	12.043	11.616	2.018	0.093	0.043
2^{16}	12.139	12.489	12.160	12.162	1.508	0.077	0.027
2^{18}	12.204	11.988	12.183	12.177	1.976	0.109	0.050
2^{20}	12.491	11.868	12.032	12.031	1.046	0.060	0.038
2^{22}	13.966	14.128	14.178	14.037	1.341	0.081	0.030
2^{24}	14.134	14.266	14.298	14.468	2.426	0.188	0.114
2^{26}	16.679	15.176	16.226	15.593	5.834	0.320	0.091
2^{28}	17.024	17.186	15.567	15.950	4.491	0.268	0.077
2^{30}	17.334	17.385	18.473	17.498	3.449	0.220	0.056
2^{32}	18.648	18.918	18.672	18.791	1.606	0.152	0.066

SID Subsum Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	40.177	40.494	39.920	40.588	39.629	39.593	39.894
2^{12}	45.876	46.656	46.941	47.104	47.011	47.560	46.421
2^{14}	52.327	52.600	52.785	52.670	53.175	52.987	53.420
2^{16}	60.349	58.958	59.428	59.895	59.650	59.236	59.190
2^{18}	65.773	66.137	66.108	66.887	65.698	66.223	66.861
2^{20}	72.454	72.586	71.827	72.017	72.892	72.144	73.838
2^{22}	79.873	79.197	78.900	78.819	79.736	78.800	80.520
2^{24}	85.493	86.787	86.069	86.752	84.986	86.469	87.442
2^{26}	92.585	92.453	92.195	92.367	92.136	92.346	91.853
2^{28}	99.119	99.165	99.031	100.282	98.818	98.578	100.012
2^{30}	104.755	105.918	105.295	106.388	105.441	105.117	104.473
2^{32}	111.694	111.609	112.343	112.971	113.046	111.414	110.797

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	39.585	39.100	39.715	39.869	0.866	0.143	0.018
2^{12}	47.672	47.461	46.413	46.911	0.972	0.182	0.016
2^{14}	52.677	52.958	52.253	52.785	0.530	0.114	0.012
2^{16}	59.867	61.144	59.124	59.684	0.844	0.211	0.024
2^{18}	65.425	66.792	65.962	66.187	0.609	0.162	0.011
2^{20}	72.315	73.640	72.396	72.611	0.699	0.210	0.017
2^{22}	79.236	80.606	79.416	79.510	0.678	0.209	0.014
2^{24}	85.511	88.646	87.340	86.549	0.975	0.346	0.024
2^{26}	91.981	93.564	91.966	92.345	0.345	0.154	0.013
2^{28}	99.970	99.058	100.214	99.425	0.559	0.198	0.009
2^{30}	105.585	106.616	104.667	105.426	0.535	0.227	0.011
2^{32}	112.100	115.110	114.670	112.575	0.976	0.443	0.023

Parallel SID Subsum Pippenger

2^b	1	2	3	4	5	6	7
2^{10}	8.219	8.589	8.118	9.024	8.006	8.036	8.180
2^{12}	11.927	12.237	11.771	11.631	11.937	10.694	11.905
2^{14}	11.950	12.050	12.560	12.290	12.202	12.014	12.596
2^{16}	12.453	12.657	12.657	12.605	12.006	12.529	12.788
2^{18}	12.474	12.532	12.517	12.502	12.627	12.553	12.720
2^{20}	14.532	14.579	13.623	14.190	14.771	14.400	13.611
2^{22}	15.078	14.909	14.391	14.635	14.771	14.391	14.834
2^{24}	15.026	16.552	14.946	16.556	14.771	15.562	16.222
2^{26}	17.762	16.855	17.690	17.612	17.635	17.616	17.521
2^{28}	17.764	17.841	18.698	18.768	16.847	17.089	17.988
2^{30}	18.277	19.478	18.331	18.907	17.053	18.867	18.848
2^{32}	20.085	20.197	19.669	20.584	20.251	19.690	20.623

2^b	8	9	10	μ	CV	SE	NPD
2^{10}	8.012	7.936	8.230	8.235	2.776	0.105	0.096
2^{12}	10.712	11.795	11.891	11.650	3.284	0.165	0.050
2^{14}	11.921	12.742	12.522	12.285	2.095	0.095	0.037
2^{16}	12.911	13.298	12.542	12.645	1.722	0.105	0.052
2^{18}	12.384	12.627	12.280	12.522	0.720	0.040	0.016
2^{20}	13.744	13.906	14.321	14.168	2.521	0.133	0.043
2^{22}	14.721	15.218	14.395	14.734	1.546	0.092	0.033
2^{24}	14.838	15.483	15.802	15.576	3.632	0.218	0.063
2^{26}	15.746	14.977	15.418	16.883	5.375	0.342	0.052
2^{28}	17.323	17.085	17.721	17.712	2.829	0.207	0.060
2^{30}	17.685	19.694	18.947	18.609	3.319	0.252	0.058
2^{32}	20.549	20.050	20.451	20.215	1.369	0.110	0.020

B.2.3 Number of point-scalar pairs

Naïve MSM

n	1	2	3	4
10	7.545	7.364	5.166	3.571
10^2	32.865	32.090	31.982	32.615
10^3	316.391	317.320	323.024	324.446
10^4	3164.395	3209.373	3203.954	3174.862
10^5	31465.012	31848.210	31717.593	31744.720
10^6	313086.363	318572.215	377057.859	324286.837
10^7	3372046.259	3149686.194	3253696.422	3245931.881

n	5	6	7	8
10	7.421	4.766	8.912	6.005
10^2	32.267	32.271	32.277	31.839
10^3	322.826	325.603	319.223	326.442
10^4	3186.938	3184.974	3220.260	3206.013
10^5	31313.074	31656.156	32409.216	31558.003
10^6	320832.842	321842.360	324621.560	320459.796
10^7	3154671.501	3107347.281	3167302.989	3160727.069

n	9	10	μ	CV	SE	NPD
10	7.127	6.112	6.399	19.926	0.501	0.393
10^2	32.341	31.661	32.221	0.814	0.112	0.020
10^3	334.000	316.457	322.573	1.296	1.741	0.035
10^4	3176.962	3415.754	3214.348	1.290	23.062	0.063
10^5	31669.121	32946.642	31832.775	1.072	153.950	0.035
10^6	315060.055	317790.398	325361.028	3.178	5860.746	0.159
10^7	3271303.378	3225091.696	3210780.467	1.957	24704.178	0.050

Trivial MSM

n	1	2	3	4
10	6.080	6.284	4.459	5.815
10^2	31.177	31.808	31.225	32.027
10^3	320.078	319.039	319.078	322.958
10^4	3199.980	3173.334	3194.046	3163.973
10^5	31545.068	31792.938	31455.227	31614.566
10^6	314482.309	320784.487	350705.514	322113.533
10^7	3251729.138	3137288.051	3183395.572	3178570.313

n	5	6	7	8
10	6.052	4.469	6.049	5.511
10^2	31.808	31.751	32.773	31.289
10^3	322.733	321.290	327.051	321.438
10^4	3159.304	3252.319	3201.159	3193.877
10^5	31780.216	31583.211	32357.458	32126.150
10^6	322879.766	322909.031	323131.965	320389.517
10^7	3123843.760	3121620.086	3152842.726	3152877.397

n	9	10	μ	CV	SE	NPD
10	6.162	5.464	5.634	9.351	0.212	0.115
10^2	32.517	31.490	31.786	1.259	0.170	0.031
10^3	320.367	321.983	321.602	0.517	0.744	0.017
10^4	3170.979	3355.465	3206.444	1.216	18.581	0.046
10^5	31610.416	32852.801	31871.805	1.080	140.459	0.031
10^6	314599.096	317331.829	322932.705	1.732	3258.073	0.086
10^7	3191127.745	3169117.573	3166241.236	0.902	12186.998	0.027

Pippenger

n	1	2	3	4
10	14.390	13.529	13.514	14.524
10^2	12.583	12.803	12.739	12.902
10^3	45.035	45.937	45.377	46.145
10^4	368.811	370.020	364.877	362.819
10^5	3715.448	3598.079	3647.033	3635.665
10^6	36449.465	37356.453	36757.956	37058.507
10^7	365811.156	372898.395	366869.858	369117.735

n	5	6	7	8
10	13.967	11.386	14.650	13.060
10^2	13.035	12.848	13.037	12.671
10^3	48.114	45.990	46.750	48.073
10^4	366.388	384.032	372.399	373.058
10^5	3664.306	3608.990	3750.886	3634.390
10^6	37294.018	37427.230	37417.986	37469.636
10^7	373042.802	377761.805	372294.735	369753.421

n	9	10	μ	CV	SE	NPD
10	14.867	13.925	13.781	5.277	0.322	0.079
10^2	13.084	12.668	12.837	1.123	0.055	0.019
10^3	45.545	47.639	46.461	2.038	0.357	0.036
10^4	362.862	399.391	372.466	2.099	3.587	0.072
10^5	3617.634	3931.141	3680.357	1.937	31.688	0.068
10^6	36401.329	36437.236	37006.982	1.071	142.635	0.013
10^7	372426.401	364867.193	370484.350	0.864	1252.736	0.020

Parallel Pippenger

n	1	2	3	4
10	3.697	3.692	4.032	3.733
10^2	3.615	3.615	3.552	3.620
10^3	12.299	12.269	12.490	12.294
10^4	97.744	97.521	97.370	99.182
10^5	974.041	949.871	950.635	964.493
10^6	9571.167	9604.564	9577.811	9599.630
10^7	94789.552	95264.083	94436.150	95899.304

n	5	6	7	8
10	3.467	3.856	4.215	3.704
10^2	3.654	3.655	3.670	3.589
10^3	12.279	12.289	12.158	13.631
10^4	97.148	98.010	98.205	96.518
10^5	956.768	1026.709	966.058	967.450
10^6	9624.442	9650.497	9697.336	9804.726
10^7	96364.059	96758.190	95377.888	94220.012

n	9	10	μ	CV	SE	NPD
10	6.892	3.686	4.097	14.216	0.317	0.682
10^2	3.675	3.562	3.621	0.945	0.014	0.015
10^3	12.398	12.365	12.447	1.971	0.134	0.095
10^4	97.413	100.234	97.935	0.795	0.339	0.023
10^5	964.252	1143.265	986.354	4.000	18.744	0.159
10^6	9567.409	9512.510	9621.009	0.609	25.859	0.019
10^7	94718.470	96932.298	95476.001	0.848	307.156	0.015

SID Pippenger

n	1	2	3	4
10	6.872	6.777	7.122	6.892
10^2	8.738	8.882	8.715	8.940
10^3	46.415	45.879	45.557	45.878
10^4	415.224	415.186	404.228	440.070
10^5	4266.475	4183.544	4118.495	4096.503
10^6	41878.927	42127.832	42250.635	41928.907
10^7	420186.191	414670.252	417302.795	424324.971

n	5	6	7	8
10	5.334	5.007	7.619	6.645
10^2	9.092	8.803	9.037	8.803
10^3	46.325	45.307	46.066	47.801
10^4	418.208	423.891	426.095	417.716
10^5	4120.317	4098.358	4222.725	4136.293
10^6	42123.195	42363.276	42268.468	42248.794
10^7	423091.534	419396.978	419553.825	427001.669

n	9	10	μ	CV	SE	NPD
10	6.806	6.340	6.541	8.998	0.252	0.165
10^2	8.965	8.827	8.880	1.159	0.040	0.024
10^3	45.777	45.640	46.064	1.020	0.220	0.038
10^4	410.445	442.717	421.378	2.243	3.868	0.051
10^5	4101.460	4381.837	4172.601	1.746	29.546	0.050
10^6	41727.154	41839.398	42075.658	0.441	68.555	0.007
10^7	423945.451	421692.603	421116.627	0.687	1152.426	0.014

Subsum Pippenger

n	1	2	3	4
10	4.612	3.795	4.335	4.804
10^2	7.217	7.175	7.066	7.235
10^3	45.442	46.023	45.641	45.599
10^4	373.626	377.063	365.069	396.743
10^5	3675.129	3706.000	3685.534	3704.845
10^6	37486.750	37720.460	37872.390	37551.515
10^7	377792.458	378076.108	379781.237	366177.865

n	5	6	7	8
10	3.666	4.115	5.668	3.753
10^2	7.294	7.391	7.381	7.171
10^3	45.869	45.820	47.986	47.689
10^4	383.714	387.175	379.731	379.527
10^5	3728.525	3628.429	3762.712	3753.455
10^6	37795.415	37894.509	37828.306	37384.808
10^7	372689.965	379205.847	384445.495	375167.598

n	9	10	μ	CV	SE	NPD
10	4.357	3.993	4.310	10.334	0.192	0.315
10^2	7.285	7.332	7.255	1.129	0.032	0.019
10^3	46.168	45.896	46.213	1.406	0.280	0.038
10^4	366.134	405.189	381.397	2.477	3.980	0.062
10^5	3702.090	3878.324	3722.504	1.252	21.175	0.042
10^6	37736.123	37068.671	37633.895	0.555	82.771	0.007
10^7	369408.167	377484.206	376022.895	1.098	1690.490	0.022

Parallel SID Pippenger

n	1	2	3	4
10	2.036	2.114	1.996	3.733
10^2	2.442	2.464	2.473	3.620
10^3	11.138	11.147	11.143	12.294
10^4	98.061	97.936	97.811	99.182
10^5	972.892	981.377	974.356	964.493
10^6	9747.726	9747.031	9716.136	9599.630
10^7	97856.816	97333.454	96317.691	96458.176

n	5	6	7	8
10	1.928	1.668	2.122	1.913
10^2	2.495	2.454	2.491	2.464
10^3	11.461	11.304	11.455	11.417
10^4	100.051	98.419	99.029	98.268
10^5	974.906	972.507	974.432	971.383
10^6	9777.230	9813.872	9816.856	10106.256
10^7	97402.985	97870.249	97246.872	97984.333

n	9	10	μ	CV	SE	NPD
10	1.845	1.907	2.126	15.118	0.183	0.756
10^2	2.453	2.558	2.591	7.940	0.115	0.397
10^3	11.125	11.251	11.373	1.993	0.111	0.081
10^4	98.740	114.924	100.242	2.929	1.645	0.146
10^5	998.897	1064.220	984.946	1.893	9.260	0.080
10^6	9981.108	9710.885	9801.673	1.043	45.731	0.031
10^7	97700.442	97643.575	97381.459	0.446	182.688	0.006

Parallel Subsum Pippenger

n	1	2	3	4
10	1.423	1.216	1.489	1.368
10^2	2.166	2.157	2.095	2.146
10^3	12.286	12.414	12.483	12.302
10^4	98.381	98.924	100.225	99.385
10^5	963.817	976.652	964.347	970.080
10^6	9669.270	9743.416	9676.051	10346.731
10^7	96601.051	97804.959	96478.952	97141.682

n	5	6	7	8
10	1.185	1.556	1.992	1.340
10^2	2.120	2.167	2.212	2.113
10^3	12.173	12.177	12.403	12.283
10^4	98.291	98.633	100.517	97.896
10^5	966.893	975.938	976.862	976.967
10^6	9747.773	9687.910	9735.452	9737.307
10^7	97420.076	97519.040	97262.295	97809.459

n	9	10	μ	CV	SE	NPD
10	1.438	1.397	1.440	9.934	0.071	0.383
10^2	2.247	2.109	2.153	1.704	0.015	0.044
10^3	12.367	12.082	12.297	0.787	0.040	0.015
10^4	99.467	110.507	100.223	2.112	1.173	0.103
10^5	970.380	1037.221	977.916	1.213	6.788	0.061
10^6	9700.458	9716.945	9776.131	1.167	64.037	0.058
10^7	97351.799	96960.233	97234.955	0.362	143.057	0.006

SID Subsum Pippenger

n	1	2	3	4
10	2.921	2.619	2.813	3.079
10^2	6.847	7.218	6.814	7.039
10^3	45.863	46.615	45.753	45.696
10^4	421.633	420.774	443.368	411.982
10^5	4166.365	4218.745	4146.321	4210.207
10^6	42516.576	42764.032	42597.985	42694.203
10^7	420591.695	426227.408	417552.571	427270.669

n	5	6	7	8
10	2.409	2.634	3.148	2.474
10^2	7.170	7.085	7.103	6.873
10^3	45.925	46.444	46.117	46.963
10^4	413.028	419.211	440.418	415.486
10^5	4151.918	4198.181	4268.675	4172.093
10^6	42729.966	42748.243	42745.685	42431.631
10^7	422275.987	423443.999	427305.200	428213.196

n	9	10	μ	CV	SE	NPD
10	2.903	2.781	2.778	7.033	0.077	0.133
10^2	7.021	7.296	7.047	1.813	0.051	0.035
10^3	46.808	46.710	46.289	0.904	0.150	0.015
10^4	416.058	451.310	425.327	2.780	4.487	0.061
10^5	4177.174	4433.394	4214.307	1.319	26.921	0.052
10^6	41710.641	42535.126	42547.409	0.468	99.927	0.005
10^7	426086.809	427974.442	424694.197	0.702	1135.943	0.008

Parallel SID Subsum Pippenger

n	1	2	3	4
10	1.137	0.906	1.068	1.047
10^2	1.969	2.061	1.986	1.959
10^3	11.230	11.136	11.448	11.158
10^4	99.280	99.105	100.402	99.310
10^5	995.634	991.938	986.053	991.427
10^6	9827.326	9924.027	9839.064	9891.683
10^7	99367.079	99721.019	99653.048	99319.770

n	5	6	7	8
10	1.012	1.061	1.006	0.969
10^2	2.016	2.006	1.986	1.988
10^3	11.333	11.164	11.201	11.318
10^4	99.457	99.085	99.987	99.345
10^5	993.073	980.621	989.537	983.487
10^6	9854.707	9905.964	9957.840	10219.744
10^7	99085.888	99725.959	97590.657	99290.787

n	9	10	μ	CV	SE	NPD
10	1.073	0.960	1.024	5.225	0.021	0.111
10^2	1.953	2.051	1.998	1.442	0.012	0.032
10^3	11.170	11.338	11.250	0.780	0.033	0.018
10^4	98.658	108.666	100.330	1.676	0.939	0.083
10^5	991.400	1009.626	991.280	0.513	2.503	0.019
10^6	9911.263	9806.790	9913.841	0.726	37.150	0.031
10^7	98052.362	98682.076	99048.864	0.570	230.066	0.007

B.2.4 Window sizes

Naïve MSM

c	1	2	3	4	5	6	7
2	319.808	314.027	321.388	320.758	324.756	315.200	325.457
3	321.006	316.675	320.413	317.757	318.342	325.856	323.407
4	312.903	320.105	316.010	316.701	319.447	314.601	315.694
6	327.521	313.367	315.220	320.386	317.062	314.840	321.888
8	326.536	313.706	320.957	314.986	317.629	317.304	317.417
10	324.989	313.312	313.730	314.331	316.112	320.892	317.693
12	313.689	313.210	315.982	314.199	314.963	319.075	317.011
16	313.673	313.153	313.569	313.981	315.771	317.051	317.024

c	8	9	10	μ	CV	SE	NPD
2	315.923	319.579	317.027	319.392	0.964	1.227	0.019
3	314.849	313.264	322.866	319.443	1.022	1.260	0.020
4	314.790	315.600	316.027	316.188	0.486	0.684	0.012
6	318.618	316.589	315.218	318.071	1.014	1.340	0.030
8	318.174	314.088	321.829	318.263	0.913	1.245	0.026
10	321.316	313.993	316.794	317.316	0.985	1.240	0.024
12	317.726	328.420	318.017	317.229	0.903	1.391	0.035
16	317.245	321.095	316.119	315.868	0.582	0.768	0.017

Trivial MSM

c	1	2	3	4	5	6	7
2	318.744	321.283	319.193	321.761	319.706	322.983	320.577
3	315.963	312.374	315.032	316.253	321.816	324.222	327.260
4	313.086	315.939	314.102	315.976	315.742	314.059	315.556
6	324.114	314.159	320.642	315.466	314.487	312.491	319.338
8	321.436	312.273	316.924	315.639	315.179	317.376	314.815
10	314.516	312.569	313.600	314.450	318.233	312.569	314.907
12	312.961	312.355	315.848	314.547	317.150	316.699	320.507
16	316.433	312.347	312.777	314.694	315.630	315.303	314.339

c	8	9	10	μ	CV	SE	NPD
2	320.993	322.898	327.669	321.581	0.559	0.813	0.019
3	313.278	322.262	317.462	318.592	1.330	1.577	0.027
4	313.768	313.161	317.293	314.868	0.392	0.448	0.008
6	318.621	319.865	316.980	317.616	0.913	1.123	0.020
8	322.243	315.254	328.258	317.940	1.140	1.493	0.032
10	317.669	313.225	315.161	314.690	0.458	0.616	0.011
12	317.085	321.118	315.819	316.409	0.665	0.898	0.015
16	317.694	320.261	316.601	315.608	0.544	0.737	0.015

Pippenger

c	1	2	3	4	5	6	7
2	106.163	111.819	109.009	112.704	108.364	110.268	109.269
3	86.086	84.493	85.693	86.522	86.953	87.757	87.172
4	67.891	67.330	67.282	67.995	67.305	67.127	71.903
6	54.219	53.264	54.711	54.052	54.218	53.034	54.644
8	46.280	44.780	45.968	44.750	44.820	45.124	47.442
10	62.604	61.244	64.050	61.322	61.741	61.838	62.200
12	103.931	102.294	104.877	102.437	103.951	107.265	104.982
16	1181.031	1159.705	1165.813	1162.131	1164.953	1168.136	1166.112

c	8	9	10	μ	CV	SE	NPD
2	109.447	108.874	112.200	109.812	1.410	0.630	0.026
3	85.175	85.653	86.305	86.181	0.883	0.309	0.018
4	67.724	67.348	68.439	68.034	1.256	0.449	0.057
6	54.462	53.312	55.378	54.129	1.055	0.234	0.023
8	46.769	44.944	46.741	45.762	1.919	0.317	0.037
10	63.773	61.373	62.258	62.240	1.196	0.312	0.029
12	106.685	105.903	105.079	104.740	1.212	0.519	0.024
16	1189.723	1177.744	1182.337	1171.769	0.747	3.199	0.015

Parallel Pippenger

c	1	2	3	4	5	6	7
2	18.994	17.471	18.279	20.108	19.042	17.994	18.350
3	20.743	15.522	14.991	16.828	15.546	16.349	15.412
4	15.117	15.587	14.925	15.065	15.062	15.155	14.967
6	10.457	10.396	10.284	10.334	10.313	10.316	11.801
8	12.075	12.249	12.273	12.273	12.292	12.328	12.303
10	19.464	19.632	19.780	19.587	19.653	19.811	19.530
12	49.182	47.640	47.941	47.734	47.700	51.305	48.016
16	611.047	605.414	607.630	600.322	605.167	600.875	605.422

c	8	9	10	μ	CV	SE	NPD
2	20.276	18.211	17.779	18.650	4.095	0.299	0.087
3	16.801	15.180	15.849	16.322	6.656	0.531	0.271
4	15.037	15.045	13.051	14.901	2.483	0.213	0.046
6	10.470	10.471	11.055	10.590	3.166	0.152	0.114
8	12.360	12.308	12.424	12.289	0.463	0.029	0.011
10	19.525	19.674	19.809	19.646	0.504	0.039	0.008
12	47.729	47.890	48.414	48.355	1.587	0.359	0.061
16	612.291	603.453	611.766	606.339	0.573	1.359	0.010

SID Pippenger

c	1	2	3	4	5	6	7
2	113.534	115.120	116.082	123.661	114.288	114.201	114.172
3	92.790	87.329	88.965	89.006	89.029	91.239	90.066
4	72.331	72.028	72.260	73.168	72.267	72.158	76.295
6	53.306	53.484	55.090	53.086	53.717	53.096	55.027
8	47.538	45.010	45.120	45.121	45.847	45.324	46.095
10	49.152	48.791	49.599	48.886	48.869	51.004	49.688
12	67.529	65.967	66.632	66.207	67.301	69.711	68.425
16	604.808	598.763	603.318	607.997	606.725	602.682	600.829

c	8	9	10	μ	CV	SE	NPD
2	115.296	115.451	116.547	115.835	1.517	0.918	0.068
3	87.946	89.624	89.758	89.575	1.251	0.495	0.036
4	72.556	72.188	72.509	72.776	1.075	0.404	0.048
6	54.192	53.175	54.996	53.917	1.349	0.266	0.022
8	46.436	45.149	45.440	45.708	1.349	0.253	0.040
10	50.318	48.797	49.396	49.450	1.136	0.232	0.031
12	68.062	68.424	67.827	67.608	1.303	0.361	0.031
16	613.546	605.835	612.531	605.703	0.598	1.498	0.013

Subsum Pippenger

c	1	2	3	4	5	6	7
2	107.318	105.959	107.218	107.911	105.731	105.746	105.872
3	88.437	83.875	83.888	85.197	86.294	90.466	87.287
4	68.125	67.044	68.171	67.556	67.298	67.662	72.033
6	54.848	53.121	55.632	53.208	53.543	54.024	56.174
8	47.336	45.011	45.080	44.953	45.127	45.359	45.518
10	51.860	51.142	51.648	51.444	51.634	52.793	51.624
12	46.444	46.079	46.374	45.936	47.054	48.012	48.309
16	44.260	45.317	46.021	45.006	44.736	44.372	44.695

c	8	9	10	μ	CV	SE	NPD
2	106.512	107.171	107.286	106.673	0.664	0.254	0.012
3	84.648	86.184	85.182	86.146	1.843	0.663	0.050
4	67.881	67.354	67.469	68.059	1.220	0.456	0.058
6	54.627	53.302	54.514	54.299	1.583	0.333	0.035
8	46.404	45.230	45.580	45.560	1.159	0.238	0.039
10	52.907	51.451	51.734	51.824	0.806	0.182	0.021
12	48.029	48.168	46.685	47.109	1.733	0.295	0.025
16	44.372	46.743	44.765	45.029	1.330	0.252	0.038

Parallel SID Pippenger

c	1	2	3	4	5	6	7
2	20.338	20.258	20.614	20.900	20.645	19.366	18.350
3	17.365	16.789	16.323	15.678	15.941	17.412	15.412
4	15.226	15.113	15.038	15.262	15.756	15.754	14.967
6	15.086	10.195	10.296	10.321	10.328	10.782	11.801
8	11.436	11.187	11.103	11.079	11.119	11.450	12.303
10	15.056	14.891	14.951	14.973	14.618	14.711	19.530
12	28.902	28.833	28.609	29.062	28.542	29.156	48.016
16	309.000	307.531	309.663	307.170	307.783	308.226	605.422

c	8	9	10	μ	CV	SE	NPD
2	19.313	20.197	20.769	20.075	3.184	0.257	0.041
3	16.871	17.534	17.109	16.643	3.869	0.241	0.054
4	15.135	15.257	15.251	15.276	1.254	0.086	0.031
6	10.444	10.368	10.690	11.031	8.747	0.474	0.368
8	11.294	11.074	11.176	11.322	2.159	0.117	0.087
10	15.056	15.024	15.099	15.391	5.379	0.463	0.269
12	29.423	28.948	29.431	30.892	11.086	1.905	0.554
16	310.794	307.078	310.717	338.338	15.788	29.679	0.789

Parallel Subsum Pippenger

c	1	2	3	4	5	6	7
2	18.424	18.485	18.617	19.341	16.782	18.988	18.863
3	16.926	16.388	16.172	16.079	16.186	17.689	16.072
4	15.007	14.796	14.658	14.803	15.085	14.889	14.540
6	13.535	10.362	10.313	10.364	10.372	10.723	14.502
8	12.419	12.270	12.318	12.263	12.312	12.160	12.338
10	16.441	16.133	16.176	16.129	16.223	16.268	16.263
12	18.821	18.796	18.742	18.713	18.136	18.861	18.740
16	23.387	24.087	24.108	24.300	23.896	23.260	23.661

c	8	9	10	μ	CV	SE	NPD
2	19.334	19.192	17.785	18.581	3.067	0.251	0.041
3	16.213	16.575	16.802	16.510	2.363	0.162	0.071
4	15.211	15.548	15.143	14.968	1.541	0.093	0.039
6	10.448	10.372	10.383	11.137	10.347	0.487	0.302
8	12.431	12.335	12.652	12.350	0.734	0.042	0.024
10	16.260	16.147	16.331	16.237	0.465	0.031	0.013
12	18.561	18.669	18.914	18.695	0.771	0.070	0.012
16	23.498	24.388	23.490	23.808	1.463	0.127	0.024

SID Subsum Pippenger

c	1	2	3	4	5	6	7
2	116.252	112.970	113.496	114.568	112.355	114.785	115.338
3	88.194	87.592	88.943	89.896	88.466	89.799	89.153
4	72.034	71.631	73.859	72.868	72.094	71.885	75.463
6	56.697	53.274	54.337	53.390	54.330	54.492	55.117
8	47.963	45.563	45.398	45.207	45.488	45.587	46.661
10	47.409	47.096	47.200	47.440	47.685	47.861	48.868
12	44.477	44.135	44.630	44.462	44.618	47.151	45.825
16	45.971	44.596	45.120	44.401	45.368	44.856	44.161

c	8	9	10	μ	CV	SE	NPD
2	114.496	114.312	114.236	114.281	0.712	0.356	0.017
3	88.003	88.986	88.215	88.725	0.711	0.241	0.013
4	72.185	72.034	71.970	72.602	1.207	0.377	0.039
6	55.190	53.349	54.515	54.469	1.346	0.330	0.041
8	46.420	45.453	47.156	46.090	1.667	0.291	0.041
10	48.424	47.227	47.974	47.718	0.944	0.182	0.024
12	45.940	45.735	45.264	45.224	1.679	0.296	0.043
16	46.054	45.573	45.720	45.182	1.229	0.210	0.019

Parallel SID Subsum Pippenger

c	1	2	3	4	5	6	7
2	20.451	19.940	20.594	20.777	20.401	21.688	20.918
3	16.600	17.295	16.488	15.951	16.008	16.914	16.730
4	15.259	14.933	14.906	15.285	15.417	15.105	16.385
6	15.231	10.302	10.318	10.460	10.246	10.276	11.074
8	11.557	11.172	11.185	11.212	11.185	11.431	11.344
10	14.556	14.284	14.377	14.354	14.287	14.331	14.305
12	17.413	17.709	17.900	17.587	17.837	17.872	17.800
16	21.382	21.406	21.592	21.200	21.700	21.541	21.081

c	8	9	10	μ	CV	SE	NPD
2	20.065	19.328	20.510	20.467	2.102	0.199	0.060
3	16.672	17.285	16.919	16.686	2.051	0.145	0.037
4	15.132	15.100	15.356	15.288	1.695	0.133	0.072
6	10.716	10.304	10.419	10.935	8.113	0.484	0.393
8	11.207	11.161	11.256	11.271	0.923	0.042	0.025
10	14.519	14.429	14.365	14.381	0.504	0.030	0.012
12	18.317	17.939	18.005	17.838	0.945	0.077	0.027
16	21.451	21.092	21.515	21.396	0.775	0.067	0.014

Appendix C

Project plan

Key Details

Date:	November 10, 2023
Name:	Aaneel Shalman Srazali
UCL ID:	19010755
Project title:	Exploring Algorithms for Multi-Scalar Multiplication on MNT Curves: Benchmarking and Analysis
Supervisors:	Dr Philipp Jovanovic, Maria Corte-Real Santos
External supervisors:	None

Aim and Objectives

Aim: To investigate and benchmark various algorithms for multi-scalar multiplication on MNT curves, contributing to the field of cryptographic research, especially in the application of recursive zk-SNARKs.

Objectives:

1. Conduct a comprehensive literature review of existing MSM algorithms, focusing on their application in BLS curves and the theoretical foundation for MNT curves.
2. Implement and benchmark several key algorithms for MSM on MNT curves, comparing their performance and efficiency.
3. Analyse the suitability of these algorithms for cryptographic applications, particularly in recursive zk-SNARKs.

4. Develop a clear understanding of the implications of MSM algorithm performance on MNT curves for blockchain technology and privacy-preserving cryptographic schemes.

Expected Outcomes/Deliverables

1. A detailed literature review document summarising findings on MSM algorithms and their applications.
2. A software implementation of multiple MSM algorithms on MNT curves.
3. A comprehensive benchmarking report comparing the performance of these algorithms, particularly on efficiency.
4. A final report detailing the findings and analysis of the efficiency of these algorithms.
5. [Optional] A prototype or software tool demonstrating the application of an optimized MSM algorithm on MNT curves.

Work Plan

To ensure the smooth flowing success of the project, I have outlined a structured plan. The plan is as follows:

1. **End of September to End of October (5 weeks):** The project began with a comprehensive study and review of literature on Zero-Knowledge Proofs. During this period, we also determined the specific area of focus for the final year project and defined the expected contributions. project.project
2. **November (4 weeks):** November will be dedicated to an extensive literature review of Multi-Scalar Multiplication (MSM) algorithms and their application on BLS curves. Simultaneously, an in-depth study into the properties of MNT curves pertinent to MSM will be undertaken. This will lay the groundwork for the initial implementation of basic MSM algorithms on MNT curves and the establishment of a benchmarking framework for subsequent performance analysis towards the end of the month.
3. **End of November to Mid-January (8 weeks):** I plan to utilise the next 8 weeks to implement advanced MSM algorithms. This phase also includes the extension of the benchmarking process and a further deep dive into MSM algorithms and potential optimisations. Concurrently, I plan on commencing the drafting of the final report.
4. **Mid-January to Mid-February (4 weeks):** In this period, the project is expected to pivot to conducting a thorough analysis of the benchmarking results. The insights

gained from this analysis will then inform further refinement of the MSM algorithm implementations, enhancing implementation based on the feedback.
5. **Mid-February to End of March (6 weeks):** As the project nears its final stages, all implementations and benchmarking will be expected to be finalised. The drafting of the final report is also expected to be completed by the end of this time, with a focus on detailed analysis and implications.
6. **April (3 weeks):** The final 3 weeks will be reserved for the final revisions and edits to the report. Additionally, preparations will be made for the project presentation and its submission, marking the culmination of the project's year-long journey.

Ethics Review

Given the technical and theoretical nature of this project, it is not expected to involve human participants, sensitive data, or ethical concerns typically associated with experimental research. However, any unforeseen ethical considerations that arise will be promptly addressed in consultation with the project supervisor.

Appendix D

Interim report

Key Details

Date:	January 19, 2024
Name:	Aaneel Shalman Srazali
UCL ID:	19010755
Initial project title:	Exploring Algorithms for Multi-Scalar Multiplication on MNT Curves: Benchmarking and Analysis
Current project title:	Exploring Algorithms for Multi-Scalar Multiplication on MNT Curves: Benchmarking and Analysis
Supervisors:	Dr Philipp Jovanovic, Maria Corte-Real Santos

Current Status

In the initial phase of my Final Year Project, my journey commenced with a study and literature review of Zero-Knowledge Proofs. This foundational work, which included an online course on Zero-Knowledge Proofs (ZK Learning) and literature reviews of research papers, was essential in providing a detailed understanding of a cutting-edge cryptographic technique, and enabled my supervisors and I to consider more specific areas of focus and contribution for my project.

Subsequently, we considered several topics within this field and decided on exploring further the topic, "Exploring Algorithms for Multi-Scalar Multiplication on MNT Curves: Benchmarking and Analysis". While there were papers out there on Multi-Scalar

Multiplication algorithms, their efficiencies and how they perform on Barreto-Lynn-Scott (BLS - a special class of elliptic curves) curves, there is a gap in benchmarking these algorithms with respect to the MNT Curves (another special class of elliptic curves), whose characteristics allow for the recursive use of zk-SNARKS in cryptographic applications.

With that motivation, I delved into studying elliptic curves and their relevance to cryptography, focusing on understanding the cycles of pairing-friendly elliptic curves. This required me to revise and further study the foundational concepts within Modular Arithmetic and Number Theory. I then proceeded to pursue further study on Multi-Scalar Multiplication (MSM), the strategies and challenges of these algorithms and finally, their specific applications on Barreto-Lynn-Scott (BLS) curves. This stage was crucial for gaining a comprehensive understanding of existing methodologies and their potential limitations. This was the point where I also began to study the unique properties of MNT curves that allowed for the recursive application of zk-SNARKS.

A significant decision in the progress of my project was the selection of the programming language Rust and the Arkworks library in Rust as the tool for implementation. This choice was made after recognising the built-in support for MNT curves as well as low-level control of the language, considering the performance-sensitive nature of the project. As a new Rustacean, I dedicated time to learning Rust which involved taking a 6-hour crash course. Soon after this, I proceeded to familiarise myself with the language by doing some exercises provided by online resources.

On the practical front, I have successfully implemented elliptic curve point addition and scalar multiplication algorithms. I have also produced a successful implementation of the Pippenger bucket method for MSM. To organise and streamline this work, I created a comprehensive Rust module, which would include various current and future functionalities and tests. This structured approach has facilitated easier development and testing but also laid a solid foundation for future enhancements and analysis.

Remaining Work

For the remaining three months, I have outlined a structured plan. The plan is as follows:

1. **Mid-January to Mid-February (4 weeks):** The immediate focus will be on implementing improvements to the Pippenger bucket method, as detailed in Section 4 of the paper available at IACR. This will be followed by the implementation of a new algorithm for MSM computation from the paper published in TCHES.
2. **Mid-February to Start of March (3 weeks):** In this period, my focus will be on finding the best parameters for running these algorithms on MNT curves. This phase will involve extensive benchmarking to assess and compare the performance of different algorithms. Concurrently, I will initiate the drafting of sections of the final report.
3. **Rest of March (3 weeks):** In this period, the focus will shift to completing the draft of the final report. The report will encapsulate all the findings, methodologies, and analyses conducted throughout the project.
4. **April (3 weeks):** The final three weeks will be reserved for revising and refining the report. This period will also be used to prepare for the project presentation and its submission.

Appendix E

Partial code listing

Our experimental suite contains 10 algorithm implementations and an implementation of basic operations on elliptic curves. Our test suite contains one test file for each implementation in the experimental suite. A full code listing would take up more than 100 pages so bearing in mind the 25-page limit, we have decided to include three algorithm implementations that best illustrate the progression from basic to advanced techniques and one test file that showcases our comprehensive testing strategy.

The full code and documentation are available at https://github.com/aaneelshalman/Multi-Scalar-Multiplication-on-MNT-Curves. As a requirement for the degree of BSc Computer Science at University College London, the partial code is listed below.

E.1 README.md

```
# Multi-Scalar-Multiplication-Algorithms-on-MNT-Curves

This repository is dedicated to the implementation of Multi-Scalar
    Multiplication (MSM) algorithms on the MNT4-298 curve using Rust. Our
     suite contains 10 algorithm implementations and an implementation of
     basic operations on elliptic curves. Our test suite contains one
    test file for each implementation in the experimental suite.

## System Requirements

```

```
To run these implementations, you will need the Rust programming
    environment set up on your machine. The code has been tested and
    confirmed to work with Rust edition "2021".

## Dependencies

- ark-mnt4-298: 0.4.0
- ark-ec: 0.4.0
- ark-ff: 0.4.0
- ark-std: 0.4.0
- rand: 0.8.5

Ensure that you have Cargo installed, as it will handle these
    dependencies automatically

## Installation instructions

1. Install Rust and Cargo using rustup.
2. Clone the repository to your local machine: "git clone https://github.
    com/aaneelshalman/Multi-Scalar-Multiplication-on-MNT-Curves"
3. Navigate to the cloned repository's root directory.
4. Run "cargo build --release" to compile the project.

## Usage

To run the main application and view the runtime outputs of the ten
    algorithm implementations:

1. Use the command "cargo run". This will execute the main.rs file, where
     the execution times of the algorithms are calculated and displayed.
2. Within main.rs, you can modify the window_size variable to adjust the
    window size, and num_points to change the number of point-scalar
    pairs used in the calculations.
3. The generate_scalar function includes a default maximum scalar value
    set to 4294967295, which is the maximum for a 32-bit unsigned integer
    . Feel free to adjust this value as needed to fit your testing
    requirements.
4. To obtain runtimes for specific stages of the Pippenger bucket method
    or to assess the additional cost of signed integer decomposition,
    uncomment the relevant timing lines in pippenger.rs and sid_pippenger
    .rs. By extension, you may also add these timing lines to any
```

```
   algorithm file to get hold of how long a specific step takes. With
   variability in variable names, look for lines similar to:

```rust
        use std::time::Instant; // This library will need to be used
        let start_partitioning = Instant::now(); // At the start of the
           step
        let duration_partitioning = start_partitioning.elapsed(); // At
           the end of the step
        println!("Partitioning took: {:?}", duration_partitioning); //
           Output upon "cargo run"
```

Testing

For testing of the algorithms:

1. Run cargo test to execute the test suites for all implemented
   algorithms. This will verify the correctness of each algorithm and
   ensure they are functioning as expected.
```

E.2 Package `msm`

Listing E.1: Trivial MSM

```rust
use ark_ff::Zero;
use ark_ec::Group;
use ark_mnt4_298::G1Projective;
use crate::operations::add_points;

// Trivial approach to Multi-Scalar Multiplication using doubling and
   addition
pub fn trivial_msm(points: &[G1Projective], scalars: &[u32]) ->
   G1Projective {
    // Ensure points and scalars have the same length
    assert_eq!(points.len(), scalars.len(), "Points and scalars must have
        the same length");

    let mut result = G1Projective::zero();

```

```
    // Iterate over each point and scalar pair
    for (i, point) in points.iter().enumerate() {
        let mut point_contribution = G1Projective::zero();
        let mut scalar = scalars[i];

        // Start with the current point to add if the LSB of the scalar
           is 1
        let mut current_point = *point;

        // Process each bit from LSB to MSB
        while scalar != 0 {
            if scalar & 1 == 1 {
                // Add the current point to the point_contribution if the
                    current bit is 1
                point_contribution = add_points(point_contribution,
                   current_point);
            }

            // Double current point regardless of bit
            current_point = current_point.double();
            scalar >>= 1; // Shift the scalar to the right for the next
               bit
        }

        // Add the contribution from this point-scalar pair to the total
           result
        result = add_points(result, point_contribution);
    }

    result
}
```

Listing E.2: Pippenger bucket method

```
use crate::operations::add_points;
use ark_ec::Group;
use ark_ff::Zero;
use ark_mnt4_298::G1Projective;
use std::collections::HashMap;
// use std::time::Instant;

// Main pippenger function
```

```rust
pub fn pippenger(points: &[G1Projective], scalars: &[u32], window_size: usize) -> G1Projective {

    // Ensure points and scalars have the same length
    assert_eq!(points.len(), scalars.len(), "Points and scalars must have the same length");

    let partitions = partition_msm(scalars, window_size);
    combine_partitioned_msm(&partitions, points, window_size)
}

pub struct MsmPartition {
    pub bit_index: usize,
    pub window_values: Vec<u32>,
}

/// Step 1: Split MSM with b-bit scalars into b/c MSMs with c-bit scalars. c == window_size
pub fn partition_msm(scalars: &[u32], window_size: usize) -> Vec<MsmPartition> {
    // let start_partitioning = Instant::now();

    // Calculate the total number of partitions based on window size
    // (32 + window_size - 1) / window_size is used so that if 32 divides window_size, num_partitions will return the divisor
    // But if 32 does not divide window_size, then num_partitions will round up the float instead of round down, the default in Rust
    let num_partitions = (32 + window_size - 1) / window_size;

    // Vector to hold information on partitions
    let mut partitions = Vec::new();

    for partition_index in 0..num_partitions {
        // Calculate the starting bit index for the current partition
        let bit_index = partition_index * window_size;

        // Collect the corresponding window values for each scalar
        let window_values: Vec<u32> = scalars.iter().map(|&scalar| {
            // Shift the scalar to right align the desired bits and mask off the rest
            (scalar >> bit_index) & ((1 << window_size) - 1)
```

```
        }).collect();

        // Push the partition information to the list of partitions
        partitions.push(MsmPartition { bit_index, window_values });
        // let duration_partitioning = start_partitioning.elapsed();
        // println!("Partitioning took: {:?}", duration_partitioning);
    }

    // Return the list of partitions, each containing its bit index and
       window values
    partitions
}

pub fn compute_msm_for_partition(partition: &MsmPartition, points: &[
   G1Projective], window_size: usize) -> G1Projective {
    // let start_bucketing = Instant::now();
    let mut buckets: HashMap<u32, Vec<usize>> = HashMap::new();
    for (index, &value) in partition.window_values.iter().enumerate() {
        if value != 0 {
            buckets.entry(value).or_insert_with(Vec::new).push(index);
        }
    }
    // let duration_bucketing = start_bucketing.elapsed();
    // println!("Bucketing took: {:?}", duration_bucketing);

    let max_scalar_value = (1 << window_size) - 1;
    let mut msm_result = G1Projective::zero();
    let mut temp = G1Projective::zero();

    // let start_subsum = Instant::now();
    for scalar_value in (1..=max_scalar_value).rev() {
        if let Some(indexes) = buckets.get(&scalar_value) {
            let sum_of_points: G1Projective = indexes.iter()
                .map(|&i| points[i])
                .fold(G1Projective::zero(), |acc, p| add_points(acc, p));
            temp = add_points(temp, sum_of_points);
        }
        msm_result = add_points(msm_result, temp);
    }
    // let duration_subsum = start_iterating.elapsed();
    // println!("Subsum accumulation took: {:?}", duration_subsum);
```

```
82
83      msm_result
84  }
85
86
87
88  // Step 3: Compute the final MSM result by combining all partitions
89  pub fn combine_partitioned_msm(partitions: &[MsmPartition], points: &[
        G1Projective], window_size: usize) -> G1Projective {
90      // Variable to store the final MSM result
91      let mut final_result = G1Projective::zero();
92
93      // Iterating over each partition in reverse to ensure doubling mimics
            scaling accurately
94      for partition in partitions.iter().rev() {
95          // Computing MSM for the current partition
96          let partition_msm = compute_msm_for_partition(partition, points,
                window_size);
97
98          // Double the final result window_size times to mimic scaling by
                2^bit_index
99          for _ in 0..window_size {
100             final_result = final_result.double();
101         }
102
103         // Adding the partition MSM to the final result
104         final_result = add_points(final_result, partition_msm);
105     }
106
107     // Returning the final combined MSM result
108     final_result
109 }
```

Listing E.3: Pippenger bucket method with parallelism, signed integer decomposition, and novel subsum accumulation

```
1  use crate::operations::add_points;
2  use ark_ec::Group;
3  use ark_ff::Zero;
4  use ark_mnt4_298::G1Projective;
5  use std::collections::BTreeMap;
6  use std::ops::Neg;
```

```rust
use std::thread;

// Main function for Pippenger with parallelism and Signed Integer Decomposition
pub fn parallel_sid_subsum_pippenger(points: &[G1Projective], scalars: &[u32], window_size: usize) -> G1Projective {
    assert_eq!(points.len(), scalars.len(), "Points and scalars must have the same length");

    let partitions = parallel_sid_subsum_partition_msm(scalars, window_size);
    let decomposed_partitions = parallel_sid_subsum_decompose_partitions(&partitions, window_size);
    parallel_sid_subsum_combine_partitioned_msm(&decomposed_partitions, points, window_size)
}

pub struct ParallelSidSubsumMsmPartition {
    pub bit_index: usize,
    pub window_values: Vec<u32>,
}

// Cloned struct for parallelism
impl Clone for ParallelSidSubsumMsmPartition {
    fn clone(&self) -> ParallelSidSubsumMsmPartition {
        ParallelSidSubsumMsmPartition {
            bit_index: self.bit_index,
            window_values: self.window_values.clone(),
        }
    }
}

// New struct adjusted for signed integer decomposition
pub struct ParallelSidSubsumMsmPartitionDecomposed {
    pub bit_index: usize,
    pub window_values: Vec<i64>,
}

// Cloned struct for parallelism
impl Clone for ParallelSidSubsumMsmPartitionDecomposed {
    fn clone(&self) -> ParallelSidSubsumMsmPartitionDecomposed {
```

```
        ParallelSidSubsumMsmPartitionDecomposed {
            bit_index: self.bit_index,
            window_values: self.window_values.clone(),
        }
    }
}

pub fn parallel_sid_subsum_partition_msm(scalars: &[u32], window_size: usize) -> Vec<ParallelSidSubsumMsmPartition> {
    let num_partitions = (32 + window_size - 1) / window_size;
    let mut partitions = Vec::new();

    for partition_index in 0..num_partitions {
        let bit_index = partition_index * window_size;
        let window_values: Vec<u32> = scalars.iter().map(|&scalar| {
            (scalar >> bit_index) & ((1 << window_size) - 1)
        }).collect();
        partitions.push(ParallelSidSubsumMsmPartition { bit_index, window_values });
    }

    partitions
}

pub fn parallel_sid_subsum_decompose_partitions(partitions: &[ParallelSidSubsumMsmPartition], window_size: usize) -> Vec<ParallelSidSubsumMsmPartitionDecomposed> {
    let base = 2u32.pow(window_size as u32);
    let threshold = base / 2;

    // Initialise decomposed partitions with the same structure but empty window values
    let mut decomposed_partitions: Vec<ParallelSidSubsumMsmPartitionDecomposed> = partitions.iter()
        .map(|p| ParallelSidSubsumMsmPartitionDecomposed {
            bit_index: p.bit_index,
            window_values: vec![0i64; p.window_values.len()], // Initialise with zeros
        })
        .collect();

```

```
76      // Append an extra partition for overflow handling
77      let last_bit_index = partitions.last().unwrap().bit_index +
            window_size;
78      decomposed_partitions.push(ParallelSidSubsumMsmPartitionDecomposed {
79          bit_index: last_bit_index,
80          window_values: vec![0i64; partitions[0].window_values.len()], //
                Initialise with zeros for overflow handling
81      });
82
83      // Iterate through each position of window values across all
            partitions, not window_values in the same bit_index
84      for i in 0..partitions[0].window_values.len() {
85          let mut carry = 0i64;
86
87          for j in 0..partitions.len() {
88              let window_value = partitions[j].window_values[i] as i64;
89              let adjusted_value = window_value + carry;
90              carry = 0; // Reset carry for the next window value
91
92              if adjusted_value >= threshold as i64 {
93                  decomposed_partitions[j].window_values[i] =
                        adjusted_value - base as i64;
94                  // Forward carry to the next partition's same position
95                  carry = 1;
96              } else {
97                  decomposed_partitions[j].window_values[i] =
                        adjusted_value;
98              }
99          }
100
101         if carry > 0 {
102             decomposed_partitions[partitions.len()].window_values[i] +=
                    carry;
103         }
104     }
105
106     decomposed_partitions
107 }
108
109 pub fn parallel_sid_subsum_compute_msm_for_partition(partition: &
        ParallelSidSubsumMsmPartitionDecomposed, points: &[G1Projective]) ->
```

```
    G1Projective {
110     let mut buckets: BTreeMap<u32, Vec<(usize, i64)>> = BTreeMap::new();
111 
112     // Add an empty bucket with index 0 as requirement for new
            parallel_subsum accumulation algorithm
113     buckets.insert(0, Vec::new());
114 
115     // Assign points to buckets based on the absolute value while keeping
            track of the original value's sign
116     for (index, &value) in partition.window_values.iter().enumerate() {
117         if value != 0 {
118             let abs_value = value.abs() as u32;
119             buckets.entry(abs_value).or_insert_with(Vec::new).push((index
                    , value));
120         }
121     }
122 
123     // Collect all the scalar values (keys of the buckets) into a vector
124     let scalars: Vec<u32> = buckets.keys().copied().collect();
125 
126     // Find the maximum difference between consecutive scalars
127     let max_diff = scalars.windows(2)
128         .map(|window| window[1] - window[0])
129         .max()
130         .unwrap_or(1) as usize;
131 
132     // Initialise tmp array of length max_diff + 1
133     let mut tmp = vec![G1Projective::zero(); max_diff + 1];
134 
135     // Use a peekable iterator to keep track of the next_scalar logic
136     let mut iter = buckets.iter().rev().peekable();
137 
138     // Iterate through the sorted buckets in reverse order
139     while let Some((&scalar, indexes)) = iter.next() {
140         let sum_of_points: G1Projective = indexes.iter()
141                 .map(|&(i, sign)| {
142                     let mut point = points[i];
143                     if sign < 0 {
144                         point = point.neg(); // Negate the point if the
                                original value was negative
145                     }
```

```
                    point
                })
                .fold(G1Projective::zero(), |acc, p| add_points(acc, p));

        tmp[0] = add_points(tmp[0], sum_of_points);

        // Peek to see if there is a next scalar and calculate the
           difference if so
        if let Some((&next_scalar_val, _)) = iter.peek() {
            let k = (scalar - next_scalar_val) as usize;
            if k >= 1 && k < tmp.len() {
                // Add the current sum to tmp[k] based on the gap to the
                   next scalar
                tmp[k] = add_points(tmp[k], tmp[0]);
            }
        }
    }

    let mut temp = G1Projective::zero();
    let mut msm_result = G1Projective::zero();

    // Subsum accumulation on tmp array
    for i in (1..=max_diff).rev() {
        temp = add_points(temp, tmp[i]);
        msm_result = add_points(msm_result, temp);
    }

    msm_result
}

pub fn parallel_sid_subsum_combine_partitioned_msm(partitions: &[
   ParallelSidSubsumMsmPartitionDecomposed], points: &[G1Projective],
   window_size: usize) -> G1Projective {
    let mut handles = Vec::new();

    // Spawn a thread for each partition, iterate through them in reverse
        to ensure doubling mimics the bit scaling process accurately
    for partition in partitions.iter().rev() {
        let partition_clone = partition.clone();
        let points_clone = points.to_vec();

```

```
182         let handle = thread::spawn(move || {
183             let msm_result =
                    parallel_sid_subsum_compute_msm_for_partition(&
                    partition_clone, &points_clone);
184             msm_result
185         });
186
187         handles.push(handle);
188     }
189
190     // Collect results from each thread and combine
191     let mut final_result = G1Projective::zero();
192     for handle in handles {
193         let partition_result = handle.join().unwrap();
194
195         // Double the final result window_size times to mimic scaling by
                2^bit_index
196         for _ in 0..window_size {
197             final_result = final_result.double();
198         }
199
200         // Adding the partition MSM to the final result
201         final_result = add_points(final_result, partition_result);
202     }
203
204     final_result
205 }
```

E.3 Package `tests`

Listing E.4: Test for Pippenger bucket method with signed integer decomposition

```
1 use msm::naive::naive_msm;
2 use msm::sid_pippenger::{SidMsmPartitionDecomposed,
    sid_decompose_partitions, sid_pippenger, SidMsmPartition,
    sid_partition_msm, sid_compute_msm_for_partition,
    sid_combine_partitioned_msm};
3 use msm::operations::add_points;
4 use ark_mnt4_298::G1Projective;
5 use ark_ff::Zero;
```

```rust
use ark_std::{test_rng, UniformRand};
use rand::{Rng, thread_rng};
use std::collections::HashMap;

#[test]
fn test_sid_pippenger_with_zero_scalars() {
    let mut rng = test_rng();
    let points = vec![G1Projective::rand(&mut rng), G1Projective::rand(&mut rng)];
    let scalars = vec![0, 0];
    let window_size = 2;
    assert_eq!(sid_pippenger(&points, &scalars, window_size), G1Projective::zero(), "Pippenger with zero scalars should return the zero point");
}

#[test]
fn test_sid_pippenger_with_all_ones_scalars() {
    let mut rng = test_rng();
    let points = vec![G1Projective::rand(&mut rng), G1Projective::rand(&mut rng)];
    let scalars = vec![1, 1];
    let window_size = 2;
    // Compare against result from naive msm
    let expected_result = naive_msm(&points, &scalars);
    assert_eq!(sid_pippenger(&points, &scalars, window_size), expected_result, "Pippenger with all ones scalars failed");
}

#[test]
fn test_sid_pippenger_with_large_scalars() {
    let mut rng = test_rng();
    let points = vec![G1Projective::rand(&mut rng)];
    let large_scalar = 1u32 << 30;
    let window_size = 2;
    // Compare against result from naive msm
    let expected_result = naive_msm(&points, &[large_scalar]);
    assert_eq!(sid_pippenger(&points, &[large_scalar], window_size), expected_result, "Pippenger with large scalar failed");
}

```

```rust
#[test]
fn test_sid_pippenger_with_empty_lists() {
    let points: Vec<G1Projective> = Vec::new();
    let scalars: Vec<u32> = Vec::new();
    let window_size = 2;
    assert_eq!(sid_pippenger(&points, &scalars, window_size),
        G1Projective::zero(), "Pippenger with empty lists should return
        the zero point");
}

#[test]
#[should_panic(expected = "Points and scalars must have the same length")
    ]
fn test_sid_pippenger_with_different_lengths() {
    let mut rng = test_rng();
    let points = vec![G1Projective::rand(&mut rng)];
    let scalars = vec![1, 2];
    let window_size = 2;
    let panic_result = sid_pippenger(&points, &scalars, window_size); //
        This should panic
    assert_eq!(panic_result, G1Projective::zero())
}

// Helper function to generate n points
fn generate_points(num_points: usize) -> Vec<G1Projective> {
    let mut rng = test_rng();
    (0..num_points).map(|_| G1Projective::rand(&mut rng)).collect()
}

// Helper function to generate n random scalars of type u32
fn generate_scalars(num_scalars: usize) -> Vec<u32> {
    let mut rng = thread_rng();
    (0..num_scalars).map(|_| rng.gen_range(0..65536)).collect()
}

#[test]
// Test for Step 1 Part 1: Number of partitions should be 32/c. c ==
    window_size
fn test_sid_partition_msm() {
    let scalars = vec![182, 255, 129];
    let window_size = 2;
```

```
77      let partitions = sid_partition_msm(&scalars, window_size);
78
79      assert_eq!(partitions.len(), 32/window_size, "Incorrect number of
            partitions");
80  }
81
82  #[test]
83  // Test for Step 1 Part 1: Testing behaviour of partition_msm when
        partitions does not divide window_size
84  fn test_sid_partition_msm_1() {
85      let scalars = vec![182, 255, 129];
86      let window_size = 3;
87      let partitions = sid_partition_msm(&scalars, window_size);
88
89      // The number of partitions should be (32/window_size).ceil() i.e. 11
            when scalars are u32 bit integeres and window_size = 3
90      assert_eq!(partitions.len(), 11, "Incorrect number of partitions");
91  }
92
93  #[test]
94  // Test for Step 1 Part 2: Bit_index should be (i * window_size)
95  fn test_sid_partition_msm_2() {
96      let scalars = vec![182, 255, 129];
97      let window_size = 2;
98      let partitions = sid_partition_msm(&scalars, window_size);
99
100     for (i, partition) in partitions.iter().enumerate() {
101         assert_eq!(partition.bit_index, i * window_size, "Incorrect
                bit_index");
102     }
103 }
104
105 #[test]
106 // Test for Step 1 Part 3: Testing correctness of window_values
107 fn test_sid_partition_msm_3() {
108     let scalars = vec![182, 255, 129];
109     let window_size = 2;
110     let partitions = sid_partition_msm(&scalars, window_size);
111
112     // Manually calculated expected window values for each scalar
113     let expected_values: Vec<Vec<u32>> = vec![
```

```
114         vec![2, 1, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  // For 182
                (10110110)
115         vec![3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  // For 255
                (11111111)
116         vec![1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  // For 129
                (10000001)
117     ];
118
119     for (i, ..) in partitions.iter().enumerate() {
120         for (j, &scalar) in scalars.iter().enumerate() {
121             let expected = expected_values[j][i];
122             println!("Exp: {:?}", expected);
123             let actual = (scalar >> (i * window_size)) & ((1 <<
                    window_size) - 1);
124             println!("Actual: {:?}", actual);
125             assert_eq!(actual, expected, "Incorrect window value for
                    scalar {} in partition {}", scalar, i);
126         }
127     }
128 }
129
130 // Test for Signed Integer Decomposition Step
131 #[test]
132 fn test_sid_decompose_partitions() {
133     let window_size = 2; // Example window size
134     // Define a partition with window values within the correct range for
            a window_size of 2
135     let partitions = vec![SidMsmPartition { bit_index: 0, window_values:
            vec![3, 1, 2, 3, 1] }]; // Adjusted values
136
137     // Perform decomposition
138     let decomposed_partitions = sid_decompose_partitions(&partitions,
            window_size);
139
140     let expected_decomposed_values = vec![vec![-1, 1, -2, -1, 1]];
141
142     // Compare decomposed window values against expected values
143     decomposed_partitions.iter().zip(expected_decomposed_values.iter()).
            for_each(|(decomposed, expected)| {
144         assert_eq!(decomposed.window_values, *expected, "Decomposition
                did not produce the expected result");
```

```
    });
}

#[test]
// Test for Step 2: Compute MSM for each partition
fn test_parallel_sid_compute_msm_for_partition() {
    let points = generate_points(10);
    let partitions = SidMsmPartition { bit_index: 0, window_values: vec![1, 0, 1, 0, 1, 0, 1, 0, 1, 0] };
    let window_size = 2;
    let decomposed_partitions = sid_decompose_partitions(&[partitions], window_size);

    // Initialise an accumulator for MSM results from each partition
    let mut total_msm_result = G1Projective::zero();

    // Iterate over each decomposed partition and compute its MSM contribution
    for decomposed_partition in decomposed_partitions.iter() {
        let msm_result = sid_compute_msm_for_partition(decomposed_partition, &points, window_size);
        // Accumulate the MSM result from each partition
        total_msm_result = add_points(total_msm_result, msm_result);
    }

    // Compare against the expected result
    let expected_result = points.iter().step_by(2).fold(G1Projective::zero(), |acc, &p| add_points(acc, p));
    assert_eq!(total_msm_result, expected_result, "MSM computation for partition failed");
}

// Test for Step 2: Bucket negative values in same bucket as positive values if absolute value is equal
#[test]
fn test_sid_compute_msm_for_partition_2() {
    // Define a decomposed partition with a mix of positive, negative, and zero window values
    let decomposed_partition = SidMsmPartitionDecomposed {
        bit_index: 0,
        window_values: vec![2, -2, 3, -3, 0, 1, -1, 0, 2, -2],
```

```
178     };
179
180     // Generate dummy points (the actual points are not relevant for
            bucket count verification)
181     let points = generate_points(10);
182     let window_size = 2;
183
184     // Utilise the parallel_sid_compute_msm_for_partition function to
            process the decomposed partition
185     let _ = sid_compute_msm_for_partition(&decomposed_partition, &points,
            window_size);
186
187     // Extract the bucketing logic for verification purposes
188     let mut buckets: HashMap<u32, Vec<usize>> = HashMap::new();
189     for (index, &value) in decomposed_partition.window_values.iter().
            enumerate() {
190         if value != 0 {
191             let abs_value = value.abs() as u32;
192             buckets.entry(abs_value).or_insert_with(Vec::new).push(index)
                    ;
193         }
194     }
195
196     let expected_bucket_count = 3;
197     let actual_bucket_count = buckets.len();
198
199     assert_eq!(actual_bucket_count, expected_bucket_count, "The number of
            buckets does not match the expected count.");
200 }
201
202 #[test]
203 // Test for Step 3: Compute the final MSM result by combining all
        partitions
204 fn test_sid_combine_msm() {
205     let points = generate_points(10);
206     let scalars: Vec<u32> = vec![0b01; 10]; // Scalars with only the
            second least significant bit set
207     let window_size = 2;
208
209     let partitions = sid_partition_msm(&scalars, window_size);
```

```
210     let decomposed_partitions = sid_decompose_partitions(&partitions,
            window_size);
211     let combined_result = sid_combine_partitioned_msm(&
            decomposed_partitions, &points, window_size);
212     // Compare against result from naive msm
213     let expected_result = naive_msm(&points, &scalars);
214     assert_eq!(combined_result, expected_result, "Combined MSM result is
            incorrect");
215 }
216
217 #[test]
218 // "Comprehensive test with 10 points"
219 fn test_sid_pippenger_algorithm() {
220     let points = generate_points(10);
221     let scalars: Vec<u32> = generate_scalars(10);
222     let window_size = 2;
223
224     let msm_result = sid_pippenger(&points, &scalars, window_size);
225     // Compare against result from naive msm
226     let expected_result = naive_msm(&points, &scalars);
227     assert_eq!(msm_result, expected_result, "Pippenger algorithm did not
            match expected result");
228 }
229
230 #[test]
231 // "Comprehensive test with 100 points"
232 fn test_sid_pippenger_algorithm_1() {
233     let points = generate_points(100);
234     let scalars: Vec<u32> = generate_scalars(100);
235     let window_size = 2;
236
237     let msm_result = sid_pippenger(&points, &scalars, window_size);
238     // Compare against result from naive msm
239     let expected_result = naive_msm(&points, &scalars);
240     assert_eq!(msm_result, expected_result, "Pippenger algorithm did not
            match expected result");
241 }
242
243 #[test]
244 // "Comprehensive test with 1000 points"
245 fn test_sid_pippenger_algorithm_2() {
```

```
246     let points = generate_points(1000);
247     let scalars: Vec<u32> = generate_scalars(1000);
248     let window_size = 2;
249 
250     let msm_result = sid_pippenger(&points, &scalars, window_size);
251     // Compare against result from naive msm
252     let expected_result = naive_msm(&points, &scalars);
253     assert_eq!(msm_result, expected_result, "Pippenger algorithm did not
            match expected result");
254 }
```