OPTIMSM: FPGA hardware accelerator for Zero-Knowledge MSM

Xander Pottier[1], Thomas de Ruijter[1] *, Jonas Bertels[1], Wouter Legiest[1], Michiel Van Beirendonck[1] and Ingrid Verbauwhede[1]

COSIC KU Leuven, Leuven, Belgium, {firstname.lastname}@esat.kuleuven.be

Abstract. The Multi-Scalar Multiplication (MSM) is the main barrier to accelerating Zero-Knowledge applications. In recent years, hardware acceleration of this algorithm on both FPGA and GPU has become a popular research topic and the subject of a multi-million dollar prize competition (ZPrize). This work presents OPTIMSM: Optimized Processing Through Iterative Multi-Scalar Multiplication. This novel accelerator focuses on the acceleration of the MSM algorithm for any Elliptic Curve (EC) by improving upon the Pippenger algorithm. A new iteration technique is introduced to decouple the required buckets from the window size, resulting in fewer EC computations for the same on-chip memory resources. Furthermore, we combine known optimizations from the literature for the first time to achieve additional latency improvements. Our enhanced MSM implementation significantly reduces computation time, achieving a speedup of up to ×12.77 compared to recent FPGA implementations. Specifically, for the BLS12-381 curve, we reduce the computation time for an MSM of size 2^{24} to 914 ms using a single compute unit on the U55C FPGA or to 231 ms using four U55C devices. These results indicate a substantial improvement in efficiency, paving the way for more scalable and efficient Zero-Knowledge proof systems.

Keywords: Multi-Scalar Multiplication · Elliptic Curve Cryptography · Hardware Acceleration · Zero-Knowledge Proof

1 Introduction

In recent years, Zero-Knowledge Proofs (ZKP) have become a popular cryptographic system in blockchain and other applications. A ZKP allows a prover to prove a statement to a verifier without revealing anything about that statement except for its truth. For example, proof systems exist to verify age without revealing the date of birth [KR17] or the originality of a blurred image without revealing the original image [KHSS22]. However, the latency of large ZKP systems is still multiple seconds, which halts many real-world applications. A substantial fraction of a ZKP's latency can be attributed to the Multi-Scalar Multiplication (MSM).

With r and q respectively the subgroup order and the field modulus of the EC curve, the MSM multiplies N scalars $k_i \in \mathbb{Z}_r$ with fixed Elliptic Curve (EC) points $P_i : (x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ and adds them together:

$$\text{MSM} = \sum_{i=0}^{N-1} k_i P_i.$$

[*] Xander Pottier and Thomas de Ruijter contributed equally to this work

Pippenger [Pip76] has described an algorithm to compute an MSM that saves many EC additions compared to other approaches for large N. Although Pippenger's algorithm is well established, parameters within it and optimizations on top of it are still an open research topic.

To accelerate the MSM, we rely on hardware acceleration, where special hardware blocks can speed up an algorithm or make it much more energy-efficient. This work uses the Alveo U55C produced by AMD and implements the optimizations for the BLS12-381 Elliptic Curve [BLS02]. However, the suggested concepts can be applied directly to any curve by replacing the EC addition module in the design.

Our Contributions

This paper improves upon the Pippenger algorithm by reducing the number of EC additions to compute an MSM. First, we reached a Pippenger window size of 19 bits to reduce the total number of windows. This is the largest window size ever implemented in hardware. We achieve this window size by splitting the MSM over multiple iterations to allow the reuse of on-chip memory resources at the cost of memory bandwidth. Second, we combined the efficiently computable endomorphism [GLV01] and the precomputation on the EC points [BGMW92] for the first time to save memory bandwidth. Combined, these optimizations reduce the required number of EC additions by at least 30% compared to recent FPGA implementations [IS24, RDQY23, Xav22].

Moreover, we suggest a new scheduler to avoid pipeline stalls during the bucket accumulation step that saves 10% of bucket accumulation cycles. In addition, we implemented the segmentation technique [Xav22] for the bucket aggregation step in hardware, removing 98% of the required bucket aggregation cycles.

Organization of the paper

First, Section 2 explains the three steps of the Pippenger algorithm. Section 3 subsequently introduces known optimizations over the Pippenger algorithm, such as the signed-digit representation, the efficiently computable endomorphism, and bucket segmentation. Next, Section 4 details how these optimizations are combined and introduces our novel iteration technique that decouples the required on-chip memory resources from the window size. Afterward, Section 5 holds a high-level overview of the data flow of the FPGA implementation. Finally, Section 6 lists our area and latency results, comparing them to state-of-the-art designs.

2 Pippenger Algorithm

Pippenger [Pip76] originally proposed an algorithm for efficiently computing a product of powers. This algorithm can be adapted for the Multi-Scalar Multiplication case, significantly improving naive algorithms [Sch, LL94]. In addition to the Pippenger algorithm, other efficient algorithms also exist, such as the Bos-Coster method [dR94] and the Strauss method [BS64]. However, according to Bernstein et al. [BDLO12], the Pippenger algorithm has the best performance of all the methods for large MSM sizes.

The main idea behind the Pippenger algorithm is to reduce the total number of EC additions by accumulating points before multiplying them with the scalars. Naively, one can calculate, e.g., $5P_0$ and $5P_1$ separately, to later compute $5P_0 \boxplus 5P_1$ ($\boxplus$ denotes the EC addition). In contrast, the Pippenger algorithm calculates $P_0 \boxplus P_1$ and multiplies the result with five. Furthermore, the Pippenger algorithm interleaves the multiplications with the different scalars and the summation of the results to reduce the total number of EC additions. The algorithm consists of three steps: bucket accumulation, bucket aggregation, and final window accumulation step.

2.1 Bucket Accumulation Step

The algorithm starts by representing the scalars k_i in binary format, where each scalar from $\mathbb{Z}_r$ is represented by $b := \lceil \log_2(r) \rceil$ bits, splitting them up into subscalars of c bits long. This creates $W := \lceil b/c \rceil$ subscalars $k_{i,w}$ per scalar k_i. Afterward, the subscalars are grouped according to their index w into so-called windows, as shown in Equation 1. Each window will be handled independently in further steps of the algorithm.

$$\sum_{i=0}^{N-1} k_i P_i = \sum_{i=0}^{N-1} \sum_{w=0}^{W-1} 2^{cw} k_{i,w} P_i = \sum_{w=0}^{W-1} 2^{cw} \sum_{i=0}^{N-1} k_{i,w} P_i \tag{1}$$

A set of memory elements, so-called buckets, accompanies a window w. Each bucket is associated with a specific value for the c-bit subscalar $k_{i,w}$. However, the bucket with index zero is discarded because a point times zero always results in the point at infinity, $\mathcal{O}$, the identity element for the EC addition. Correspondingly, $2^c - 1$ buckets B_s^w, $0 < s < 2^c$, are associated with each window w.

Each point will be added to a bucket in each window separately. The value of the corresponding subscalar $k_{i,w}$ determines which bucket the point is added to. A bucket holds an EC point, initialized by the point at infinity $\mathcal{O}$. Points are accumulated into the buckets, i.e., adding to a bucket is equivalent to taking the value out of the bucket and adding a new point to it. The resulting EC point is stored in the bucket. This algorithm step performs

$$W \cdot N$$

EC additions, with N the number of points in the MSM. An algorithmic description of this step can be found in Algorithm 1.

2.2 Bucket Aggregation Step

Essentially, the previous step accumulated the points that must be multiplied by the same subscalar. Next, the bucket aggregation step interleaves two operations. First, the multiplication with the bucket subscalar will be performed efficiently for each bucket. At the same time, all multiplied buckets will be accumulated into a single sum G^w. This step has to be performed for each window separately.

Given the collection of $2^c - 1$ buckets B_s^w, each storing a single EC point, the bucket aggregation step calculates the following equation:

$$G^w = \sum_{s=1}^{2^c-1} s B_s^w .$$

This equation can be computed efficiently in

$$W \cdot 2 \cdot (2^c - 1)$$

EC additions for all windows combined, as shown in Algorithm 1. The number of EC additions is typically much lower in the bucket aggregation step compared to the bucket accumulation step, as N is usually much larger than 2^c.

2.3 Final Window Accumulation

After adding the buckets together in each window, the final window accumulation can be performed. In this step, the results of all windows are correctly added together to get the final solution of the MSM. Algorithm 1 shows how this can be calculated efficiently. This last step takes

$$(W-1)\cdot(c+1)$$

EC additions, which are negligible compared to the previous two steps.

2.4 Total Complexity

The total number of Elliptic Curve additions of the complete Pippenger algorithm is given in Equation 2.

$$W\cdot(N+2\cdot(2^c-1))+(W-1)\cdot(c+1) \tag{2}$$

The window size c is an important trade-off parameter. In realistic implementations, the window size is as large as possible to reduce the overall complexity. The drawback of increasing c is the exponential growth of the number of buckets that must be stored on-chip and the computational cost of the bucket aggregation step.

Algorithm 1 The Pippenger Algorithm

```
function BucketAccumulate($P$, $k$)
    $B \leftarrow \mathcal{O}$                                    ▷ $B$ is the list of all Buckets
    for $i = 0$ to $N-1$ do
        for $w = 0$ to $W-1$ do
            $s \leftarrow \lfloor k_i/2^{c\cdot w} \rfloor \bmod 2^c$
            $B_s^w \leftarrow B_s^w \boxplus P_i$                  ▷ $B_s^w$ is bucket $s$ of window $w$
    return $B$

function BucketAggregate($B$)
    $G \leftarrow \mathcal{O}$              ▷ $G$ is the list of the aggregation results for each window
    for $w = 0$ to $W-1$ do
        $acc \leftarrow \mathcal{O}$
        for $s = 2^c - 1$ to 1 do
            $acc \leftarrow acc \boxplus B_s^w$
            $G^w \leftarrow G^w \boxplus acc$           ▷ $G^w$ is the aggregation result of window $w$
    return $G$

function FinalWindowAccumulate($G$)
    $MSM \leftarrow \mathcal{O}$
    for $w = W-1$ to 1 do
        $MSM \leftarrow MSM \boxplus G^w$
        for $i = 1$ to $c$ do
            $MSM \leftarrow MSM \boxplus MSM$
    $MSM \leftarrow MSM \boxplus G^0$
    return $MSM$
```

3 Optimizations over the Pippenger algorithm

After the introduction of the Pippenger algorithm, numerous optimizations have been found in the literature that improve on top of the original algorithm. This section discusses the most important optimizations, including their changes to the overall complexity formula given in Equation 2.

3.1 Signed-Digit Representation

The signed-digit representation [Col26] is used in the MSM algorithm to reduce the number of buckets per bucket set without changing the number of windows [Gut20]. It relies on the property that it is inexpensive to negate EC points: to compute the negative of a point, only the sign of the y-coordinate has to be inverted, adding almost no extra overhead. Due to this minimal extra work, the signed-digit representation is commonly implemented on FPGA [IS24, RDQY23].

The core idea is that when the subscalar for a window is larger than or equal to 2^{c-1}, it can be represented by $2^c - \zeta$ with ζ a $c-1$-bit integer. The negative of the point will then be added to the bucket with the number ζ and the term 2^c is corrected for in the next window by increasing that subscalar by one.

The signed-digit representation reduces the number of buckets in each window from $2^c - 1$ to 2^{c-1}, resulting in lower memory usage. Furthermore, it reduces the complexity of the bucket aggregation step, resulting in a total complexity of

$$W \cdot (N + 2 \cdot 2^{c-1}) + (W-1) \cdot (c+1). \tag{3}$$

3.2 BGMW Method

As the EC points in the MSM are constant, precomputation on these points can help reduce the number of EC additions in the bucket aggregation step. The BGMW method was suggested for MSM in [BGMW92] but has not been implemented on FPGA before, to the best of our knowledge. It suggests precomputing $2^{cl}P_i$ for each point P_i, for l in a set of carefully chosen values. This allows us to reduce the total number of bucket sets. Instead of adding point P_i to bucket set B^w, according to the w-th section of the scalar, the precomputed point $2^{cl}P_i$ can now be added to bucket set B^{w-l}. No more points require bucket set B^w, thus it is removed.

We drove this idea to its maximal potential by calculating multiples of the points for each window. This requires more off-chip storage and increased memory bandwidth compared to the original method. However, in this case, all parts of the scalar can be used to add multiples of the original point to the first bucket set. Equation 4 shows the rewritten MSM formula, with $P_{i,w}$ the precomputed multiple of the point P_i for window w.

$$\sum_{i=0}^{N-1} k_i P_i = \sum_{w=0}^{W-1} \sum_{i=0}^{N-1} k_{i,w}(2^{cw}P_i) = \sum_{w=0}^{W-1} \sum_{i=0}^{N-1} k_{i,w}P_{i,w} \tag{4}$$

As only one bucket set is kept, the bucket aggregation only has to be applied once. Furthermore, no more final window accumulation has to be performed. This reduces the total complexity of the original Pippenger algorithm to

$$W \cdot N + 2 \cdot (2^c - 1). \tag{5}$$

3.3 Efficiently Computable Endomorphism

Gallant, Lambert, and Vanstone [GLV01] describe accelerating an EC point multiplication using efficiently computable endomorphisms. This optimization has been implemented before on FPGA [HM09]. An endomorphism of an Elliptic Curve E, defined over $\mathbb{F}_q$, is a map ϕ that maps E to itself. An example is the point multiplication with any scalar $k \in \mathbb{Z}_r$, for which the map $\phi : E \to E$ is defined by $P \to kP$.

They describe in example 4 of [GLV01] that there exist efficiently computable endomorphisms for Elliptic Curves of the form

$$E : y^2 = x^3 + b \quad \text{defined over } \mathbb{F}_q \tag{6}$$

when $q \equiv 1 \pmod 3$. This is the case for all BLS and BN curves [BLS02, PJNB10] and is defined as follows:

Given E that satisfies Equation 6, define $\alpha \in \mathbb{F}_q$ as an element of order 3 and λ as an integer satisfying the equation $\lambda^2 + \lambda \equiv -1 \pmod r$, the endomorphism is defined as

$$P \to \lambda P : (x, y) \to (\alpha x, y). \tag{7}$$

It is thus possible to compute a specific multiple of a point with only one modular multiplication.

This property can be used to accelerate the Pippenger algorithm. Given a point multiplication $k_i P_i$, where k_i is b bits long, it is possible to rewrite the scalar k_i to the form $k_i = k_{1,i} + k_{2,i}\lambda$ where $k_{1,i}$ and $k_{2,i}$ are each approximately $b/2$ bits long. By splitting the scalar, there are now two point multiplications of half the original size: $k_{1,i}P_i \boxplus k_{2,i}\lambda P_i$, where λP_i can be efficiently computed due to the endomorphism property given in Equation 7. Rewriting each scalar of the MSM algorithm results in Equation 8.

$$\sum_{i=0}^{N-1} k_i P_i = \sum_{i=0}^{N-1} (k_{1,i} + k_{2,i}\lambda) P_i = \sum_{i=0}^{N-1} \sum_{w=0}^{\lceil b/2c \rceil - 1} 2^{cw} (k_{1,i,w} + k_{2,i,w}\lambda) P_i \tag{8}$$

As a result, the bucket accumulation step has twice as many points but the number of bucket sets is also reduced by half. This reduces the complexity of the bucket aggregation step and the number of buckets that must be stored, similar to the improvement achieved with the BGMW method. However, no precomputed points must be stored or used, as the endomorphism is cheap to compute. The overall complexity of the original Pippenger algorithm is reduced to Equation 9.

$$\left\lceil \frac{b/2}{c} \right\rceil \cdot (2N + 2 \cdot (2^c - 1)) + \left(\left\lceil \frac{b/2}{c} \right\rceil - 1 \right) \cdot (c + 1) \tag{9}$$

3.4 Bucket Segmentation

The last two optimizations that will be discussed do not reduce the theoretical complexity of the Pippenger algorithm but rather try to minimize the pipeline stalls when implementing the algorithm in hardware.

Xavier [Xav22] proposed a variant to the bucket aggregation step to reduce pipeline stalls on the FPGA, called bucket segmentation. As shown in line 14 of Algorithm 1, the G^w value can only be updated when the *acc* value is ready. Since the hardware implementation of the EC adder is pipelined, this will take multiple clock cycles. During this time, there are no other computations that can start, resulting in many stalls and overall longer latency. The bucket segmentation algorithm tries to minimize these stalls by breaking up the aggregation step into M smaller segments, that each can run in parallel:

$$\begin{aligned} tot = \sum_{s=1}^{2^c-1} sB_s &= \sum_{m=0}^{M-1} \sum_{i=0}^{2^c/M-1} (i + 2^c \frac{m}{M}) B_{i+2^c m/M} \\ &= \underbrace{\sum_{m=0}^{M-1} \sum_{i=0}^{2^c/M-1} iB_{i+2^c m/M}}_{\text{Segmented Aggregation}} \boxplus \underbrace{\sum_{m=0}^{M-1} 2^c \frac{m}{M} \sum_{i=0}^{2^c/M-1} B_{i+2^c m/M}}_{\text{Overhead}} . \end{aligned}$$

The segmented aggregation steps require $2M$ fewer point additions than the classic aggregation step. However, the overhead results in $3M$ additional point additions and $c - \log_2(M)$ point doublings. Although the segmented variant introduces a few more EC additions, the number of saved pipeline stalls is much greater, reducing the overall latency.

3.5 Scheduling

Where bucket segmentation addresses known pipeline stalls during the bucket aggregation step, the bucket accumulation experiences stalls that are not known beforehand. During the bucket accumulation step, pipeline stalls occur when multiple points have to be added to the same bucket in quick succession, which is known as a collision. A point that needs to be added to a recently used bucket has to wait until the previous point is added. These collisions cannot be predicted as the scalars determining which bucket a point should be added to, are uniformly random distributed and only known at runtime. If the hardware fails to detect a collision, a new fetch of the old value will be made, and the newest point will be added to it and stored in the bucket. However, the pipeline was already computing with the old value to add a different point, and that corresponding store into the bucket will be overwritten.

Therefore, the hardware must check whether it is not fetching a bucket already used in the pipeline while executing. The simplest way of dealing with this situation is to wait until the first addition to the bucket is completed before the next one is started. This stalls the hardware pipeline, and no useful operation is started. By using a Markov chain [Gag17], it is possible to statistically calculate the fraction of time the pipeline will be stalled on average. This shows that 11.1% of cycles will be lost for a pipeline depth of 128 cycles for the EC adder and 2^{16} independent buckets. As these stalls deteriorate the performance significantly, alternatives are necessary.

Aasaraai et al. [ABC+22] describe a delayed scheduler to schedule the points and avoid stalling the pipeline as much as possible. The scheduler provides an additional queue where points that collide can be stored. At the end of the bucket accumulation step, when all points are handled and put through the pipeline or collided and stored in the queue, this separate queue is handled. However, new iterations through this queue should be made when new collisions occur. Moreover, the amount of memory that must be provided for this queue poses a problem. In the worst case, almost all points will end up in this queue, which is not feasible to store. Therefore, the main process may have to halt in the middle of execution when this queue has filled up, requiring careful and complex state control. Although this scheduler reduces the fraction of pipeline stalls below 1%, the implementation is complex. Therefore, an alternative, simpler scheduler will be proposed in Subsection 4.2 that achieves comparable results.

4 Reducing Computational Complexity

When implementing the Pippenger algorithm, the window size c is an important trade-off parameter, influencing both the on-chip memory usage and the speed of the implementation. In terms of memory, c determines the number of buckets, $2^c - 1$, that must be stored on the FPGA. In terms of speed, c balances the cost in EC additions between bucket accumulation (inversely proportional in c) and bucket aggregation (exponential in c):

$$\underbrace{\left\lceil\frac{b}{c}\right\rceil\cdot N}_{\text{bucket accumulation}} + \underbrace{\left\lceil\frac{b}{c}\right\rceil\cdot 2\cdot(2^c-1)}_{\text{bucket aggregation}} + \underbrace{\left(\left\lceil\frac{b}{c}\right\rceil-1\right)\cdot(c+1)}_{\text{final window accumulation}} .$$

Figure 1 illustrates the required number of EC additions, including pipeline stalls, for different variants of the Pippenger algorithm with $N = 2^{24}$ as a function of the window size. The data points indicated with triangles denote the original Pippenger algorithm without any optimizations, with complexity given by Equation 2. The exponential growth in EC additions for higher window sizes is due to the bucket aggregation step taking exponentially longer. However, this step is only dominant over the bucket accumulation for large values of c. The number of EC additions decreases whenever c reaches a threshold value that decreases the number of windows by one. Choosing the correct window size can speed up the Pippenger algorithm by more than a factor of two.

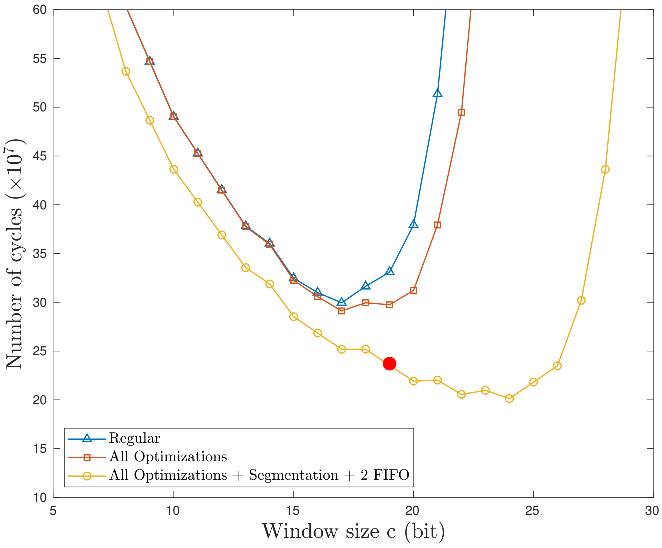

Figure 1: Number of cycles for different window sizes for the Pippenger algorithm of size 2^{24}. The red dot indicates the window size implemented in this work.

4.1 Combining The Optimizations

In this work, the precomputation on the points and the efficiently computable endomorphism are for the first time combined to accelerate an MSM. Using the precomputation optimization, multiples of the points are calculated so that it is possible to reduce the total number of bucket sets to one. Each multiple of the point still has to be added to a bucket corresponding to the subscalar, but the bucket aggregation step only has to be applied to one bucket set. Note that the endomorphism property reduces the number of bucket sets by half by splitting the scalar into two smaller scalars. If precomputation reduces the number of bucket sets to one, the endomorphism property can no longer halve the number of bucket sets, hence the original benefit is lost.

However, we found that the endomorphism property can still be exploited to reduce the precomputation required for each point. Therefore, the necessary storage and bandwidth of the memory interface are also reduced. Example 1 illustrates how both optimizations can be combined.

Example 1. Assume the scalar corresponding to the point P is split into four windows of size c. The points 2^cP, $2^{2c}P$, $2^{3c}P$ are precomputed to reduce everything to one bucket set without exploiting the efficiently computable endomorphism. However, when first applying the endomorphism property, the scalar is split into two smaller scalars. Afterward, both can be split into two windows of size c. When precomputing the values to reduce the number of bucket sets to one, only 2^cP and $2^cP'$ are necessary, with P' the efficiently computable endomorphism of point P. This can be taken one step further. Note that $2^cP' = 2^c\lambda P = \lambda 2^cP$. Therefore, it is possible only to compute 2^cP and calculate $2^cP'$ on the fly as it is the efficiently computable endomorphism of 2^cP.

Implementing the endomorphism property can still be advantageous, as it reduces the total amount of required precomputation by half at the cost of a single modular multiplication to compute the endomorphism of a point on the fly. This further implies that the required memory bandwidth is reduced by half.

After combining these two optimizations, the signed-digit representation can still reduce the window size by one bit, by allowing negative subscalars. The minus sign of the subscalar is absorbed by the point, as the negative of a point is easy to compute.

By combining all three optimizations, the number of EC additions, including pipeline stalls, in function of the window size c is shown with squares in Figure 1. Note that the optimizations reduce the number of EC additions in the bucket aggregation step, which is only dominant for large values of c. Therefore, these optimizations allow larger values of c to be reached before the aggregation step becomes dominant. Moreover, the required on-chip storage was reduced by a factor equal to the number of windows, as only a single bucket set is required. Again, this allows for larger windows, corresponding to higher values of c, showing fewer required EC additions.

4.2 Minimizing Pipeline Stalls

In Subsection 3.5, we showed that the pipeline stalls on average 11.1% of the time during the bucket accumulation step. Aasaraai et al. [ABC+22] suggested to provide an additional queue to store colliding points. Note that this queue would fill up very quickly when many collisions occur, necessitating large queues or special control logic to empty the queues when they are full. Furthermore, a cycle is lost for every colliding point to move it into the extra queue. This scheduler limits the average fraction of cycles lost to pipeline stalls during the bucket accumulation step to less than 1%. The exact number of remaining pipeline stalls will depend on the size of the additional queue.

We suggest approaching the scheduling task in a different, simpler way, providing multiple First In First Out (FIFO) queues where points imported into the FPGA can be stored and wait to be used. In every cycle, a single point will be fetched from one of the queues if it is not blocked by collisions. Points that are not taken stay in the queues, waiting to be chosen when possible. Notice that no stalls will be introduced as long as at least one point is available. Using a Markov chain to analyze the average number of stalls when using two queues, a pipeline depth of 128 cycles, and 2^{16} independent buckets, we find that 0.7% of cycles will be lost. Similar improvements will be seen when adding more than one extra queue. Hence, this solution can be scaled to trade pipeline stalls for design complexity and area.

In addition to the alternative scheduler, the bucket segmentation technique was implemented. This technique reduces the number of pipeline stalls during the bucket aggregation step. As the implemented EC adder has a pipeline depth of 128 cycles and two operations are available for each segment, $M = 64$ segments were used.

Taking the bucket segmentation optimization and the improved scheduler into account, the data points indicated with circles in Figure 1 show the number of EC additions required, including pipeline stalls, for an EC adder with a pipeline depth of 128 cycles.

4.3 Implemented Window Size

We chose to implement the Pippenger algorithm with a window size of $c = 19$ bits. This is a large window size compared to previous works with window sizes of 12 [Xav22] or 13 [RDQY23] bits. This larger window size reduces the required EC additions and the total latency, as highlighted in Figure 1. However, it requires a factor of 2^6 more on-chip memory than a window size of 13 bits without the previously discussed optimizations.

By exploiting precomputations and the efficiently computable endomorphism, all points are added to one bucket set according to the subscalar. Therefore, a larger window size than previous works can be achieved as only one set of buckets must be stored instead of one for each window.

We decided to implement 2^{16} buckets on one U55C FPGA from AMD [Xila]. This is a conservative number of buckets that maximizes the usage of available memory resources while avoiding congestion. With 2^{16} buckets, it is possible to implement windows of 16 bits, or 17 bits if the signed-digit representation is used. However, windows of 19 bits were previously suggested. A new iteration technique is suggested to fill the gap between the number of implemented buckets and the chosen window size.

4.4 Iterative Multi-Scalar Multiplication

To cover the difference between the 2^{16} available buckets and the window size of 19 bits, we propose a new technique that decouples the chosen window size from the number of implemented buckets. This novel technique executes the MSM algorithm in I independent iterations, reusing the buckets I times. During each iteration, only a subset of the point-subscalar pairs are processed based on the subscalar value. In a straightforward way, this subset is determined by the most significant bits of the subscalar.

Using this technique, it is possible to execute the Pippenger algorithm with a window size c with $2^c/I$ physical memory elements, instead of the traditional 2^c. Additionally, combining this approach with the signed-digit representation further reduces the number of buckets to $2^{c-1}/I$. This allows us to implement a larger window size for the same number of on-chip resources, reducing the required number of EC additions

Furthermore, splitting the MSM into iterations introduces a more efficient way to parallelize the execution over multiple compute units. Naively, the MSM of size N could be parallelized by dividing the problem into I smaller MSM computations of size N/I. However, I bucket aggregation steps of 2^c buckets must be performed using this naive technique. In contrast, our proposed iteration technique only requires I bucket aggregation steps of $2^c/I$ buckets.

However, the proposed iteration technique has the following limitation: When I iterations reuse the buckets, only one out of every I point-subscalar pairs will be scheduled on average in every iteration. To ensure that the EC adders can be used without interruption, the memory bandwidth has to be high enough to provide points and scalars at a sufficient rate.

In this work, we implement $I = 4$. As a consequence, we only require $2^{19-1}/4 = 2^{16}$ physical buckets. The implemented distribution of the subscalars over the iterations is as follows:

$$\begin{aligned}
&\text{Iteration 0: subscalar} \in [1, 2^{16}]\\
&\text{Iteration 1: subscalar} \in (2^{16}, 2^{17}]\\
&\text{Iteration 2: subscalar} \in (2^{17}, 2^{17} + 2^{16}]\\
&\text{Iteration 3: subscalar} \in (2^{17} + 2^{16}, 2^{18}].
\end{aligned}$$

Using multiple iterations has a further effect on the bucket aggregation step. Depending on the current iteration, a different offset has to be added to the aggregation result. For example, in the second iteration, the points in bucket $B_{i+2^{16}}$ are multiplied with the constant i, afterward an additional term of 2^{16} times the sum of all buckets has to be added to find the correct result. Equation 10 indicates which multiple has to be added to the result of the aggregation step for each iteration. Note that this second term, the sum of all buckets, is already calculated during the aggregation step. The overhead amounts to $I + c - \log_2(I)$ EC additions and is negligible compared to the benefits we gain by implementing a larger window size.

$$\begin{aligned}
tot = \sum_{i=0}^{2^{18}-1} iB_i = \sum_{i=0}^{2^{16}-1} iB_i & & \text{Iteration 0}\\
\boxplus \sum_{i=0}^{2^{16}-1} iB_{i+2^{16}} \boxplus 2^{16} \sum_{i=0}^{2^{16}-1} B_{i+2^{16}} & & \text{Iteration 1}\\
\boxplus \sum_{i=0}^{2^{16}-1} iB_{i+2\cdot 2^{16}} \boxplus 2 \cdot 2^{16} \sum_{i=0}^{2^{16}-1} B_{i+2\cdot 2^{16}} & & \text{Iteration 2}\\
\boxplus \sum_{i=0}^{2^{16}-1} iB_{i+3\cdot 2^{16}} \boxplus 3 \cdot 2^{16} \sum_{i=0}^{2^{16}-1} B_{i+3\cdot 2^{16}} & & \text{Iteration 3}
\end{aligned} \tag{10}$$

4.5 Optimizing Iteration Distribution

To achieve the most optimal speedup when parallelizing the iterations over multiple compute units, each iteration must handle the same number of point-subscalar pairs. Unfortunately, this is not the case due to the endomorphism property that is used. To calculate the 19-bit subscalars, the original 255-bit scalar is first split into a 130-bit and 128-bit scalar to use the efficiently computable endomorphism. Further, these scalars are each split into seven 19-bit subscalars.

Note that 130 (128) is not a multiple of 19. This will cause the highest subscalar to always start with three (five) consecutive zeros. Therefore, the most significant subscalar will always be in the range $[1, 2^{16}]$ and the corresponding point-subscalar pair will always be scheduled in the first iteration. As a result, the first iteration will on average handle more points and have a significantly higher computation time than the other iterations. To create a balanced distribution, we suggest two solutions.

4.5.1 First Solution: Window Shifting

The first solution can be best understood with a simple example. Given a 13-bit scalar that will be split into windows of 4 bits. Unfortunately, 13 is not a multiple of 4, which will cause the highest subscalar to consist of three zeros followed by the highest bit of the scalar. This results in the skewed distribution of the subscalars over the iterations, as seen in Figure 2.

The technique we propose is visualized in Figure 3. This technique overlaps parts of the windows to make the most significant bit for the three highest windows zero while not changing the original scalar. The signed-digit representation algorithm will remove this zero, leaving the remaining bits of the subscalar to determine the distribution over the iterations.

Remark that different points must be precomputed to shift everything down to one bucket set compared to the original scenario. In the original scenario in Figure 2, points $2^4P, 2^8P$, and $2^{12}P$ were required, while the new technique in Figure 3 requires points $2^4P, 2^7P$ and $2^{10}P$.

This technique can be performed on the 130- and 128-bit scalars with 19-bit windows. Unfortunately, in this case, the most significant bits of the scalar are skewed due to the algorithm used to generate the 130- and 128-bit scalars. Therefore, the skew in the distribution is not completely removed, even after applying the Window Shifting solution.

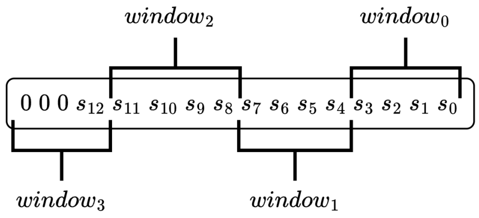
$window_2$
$window_0$
0 0 0 s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
$window_3$
$window_1$

Figure 2: Naive technique

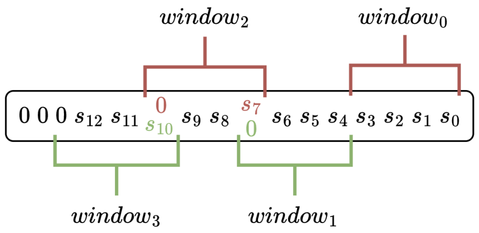
$window_2$
$window_0$
0 0 0 s_{12} s_{11} 0 s_{10} s_9 s_8 s_7 0 s_6 s_5 s_4 s_3 s_2 s_1 s_0
$window_3$
$window_1$

Figure 3: Window Shifting technique

4.5.2 Second Solution: Distribution With Offset

The second solution we propose will eliminate the skewed distribution completely by using additional precomputation. This technique exploits the fact that the three most significant bits are always zero, hence we can hardcode them to different values. This forces the most significant point-subscalar pairs to be scheduled in specific iterations and ensures an equal distribution. By tampering with this subscalar, a fixed offset on the MSM will be introduced. However, as both the adjustments and the points are known before starting the MSM, it is possible to precompute the offset on the complete result. The offset is a single EC point that must be subtracted from the result of the MSM to find the correct result.

Equation 11 calculates the introduced offset after hardcoding the three most significant bits of the most significant subscalar according to Table 1. The most significant bit will be used in the signed-digit representation to determine whether the two-complement of the subscalar has to be taken. The two other bits determine in which iteration the point-subscalar pair is scheduled.

$$\text{Offset} = (\lambda + 3) \cdot 2^{130} \cdot \sum_{i=0}^{N/2-1} P_{2\cdot i+1} \boxplus 2 \cdot 2^{130} \cdot \sum_{i=0}^{N/2-1} P_{2\cdot i} \tag{11}$$

Table 1: Distribution with offset technique by hardcoding the most significant bits of the most significant subscalar

	$P_{2 \cdot i}$	$P_{2 \cdot i+1}$
Endo	000	001
Not Endo	010	011

4.5.3 Comparison

Table 2 shows the distribution of random scalars over the four iterations. It is clear that without applying one of the discussed solutions, the first iteration creates a bottleneck if the iterations were to be parallelized over multiple compute units. The Window Shifting technique improves the distribution but fails to eliminate the skew completely. On the contrary, Distribution With Offset achieves an ideal distribution and is desired when the four iterations are parallelized over four compute units. The drawback of this technique is the static offset that has to be precomputed and corrected for in software. We chose to implement the latter technique as it results in the ideal distribution.

Table 2: Comparison of different point-subscalar pair distribution techniques. The table indicates the fraction of point-subscalar pairs scheduled in an iteration. This fraction is equivalent to the fraction of time of the total computation an iteration takes. When parallelizing the iterations over multiple compute units, the bottleneck will be the one with the highest fraction.

	Naive	Window Shifting	Distribution With Offset
Iteration 0	35.8 %	30.1 %	25 %
Iteration 1	21.4 %	26.7 %	25 %
Iteration 2	21.4 %	21.8 %	25 %
Iteration 3	21.4 %	21.4 %	25 %

5 Hardware Architecture

In this section, the complete data flow of our MSM implementation will be explained based on the high-level diagram in Figure 4.

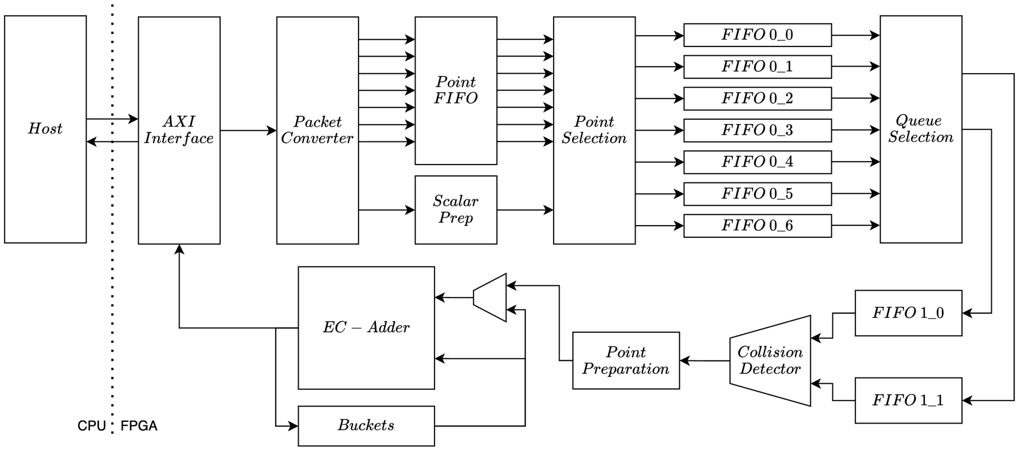

Figure 4: Complete overview of the MSM module

Initially, each point, its six precomputed multiples, and the corresponding scalar start in the DRAM memory, where the software stored them. Using a custom memory interface based on the AXI protocol [Xilb], the points and scalars are read out as AXI streams of 4096 bits in bursts of 128 packets. The *Packet Converter* module then collects these AXI streams. This module assigns the correct bits to the *Point FIFO* and *Scalar Preparation* modules. The *Point FIFO* module consists of seven FIFO blocks, storing the point and the six precomputed multiples until the *Scalar Preparation* module is ready. The *Scalar Preparation* module converts the 255-bit scalars into fourteen subscalars, according to Figure 5.

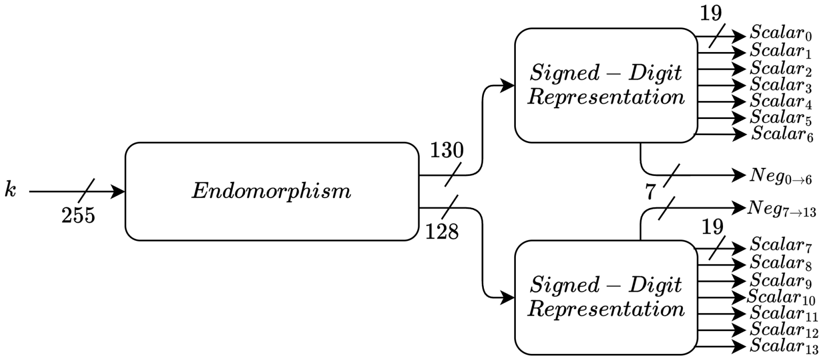

Figure 5: High-level overview of the *Scalar Preparation* module

Gallant et al. [GLV01] proposed using the extended Euclidean algorithm to calculate the decomposition of k. However, Chao [Cha] suggests using parts of the Barrett modular multiplication algorithm [Xav22] to find k_1 and k_2 more easily. Therefore, we thoroughly optimized the Barrett modular reduction for the *Endomorphism* submodule. Eventually, this submodule is implemented with only 1850 LUTs and 37 DSP blocks.

After creating the subscalars, they are recombined with their respective points from the

Point FIFO module and sent to the *Point Selection* module. This module assigns the seven point-subscalar pairs to their corresponding FIFO queues or discards them if they are not handled in this iteration. Next, the *Queue Selection* module schedules which queues should be read and passes at most one point-subscalar pair to each FIFO queue in the next module. This scheduler prioritizes point-subscalar pairs from almost full queues.

Afterward, the *Collision Detector* module selects a point-subscalar pair from one of the two FIFOs, depending on potential collisions in the pipeline. The selected pair is sent to the *Point Preparation* module. This module prepares the point for the Elliptic Curve adder by computing the endomorphism and negative, as displayed in Figure 6.

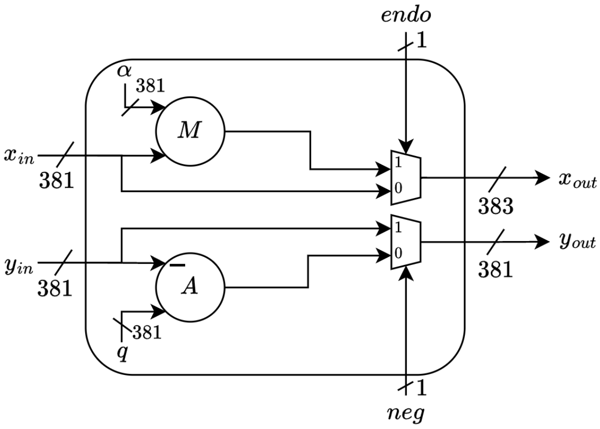

Figure 6: Overview *Point Preparation* Module

All previous modules schedule and prepare the point-subscalar pairs for the bucket accumulation step. In this step, the point from the *Point Preparation* module is used as the first input of the EC adder. Meanwhile, the subscalar linked to the point will be used as a memory address in the *Buckets* module to read the correct bucket. The bucket value is used as the second input for the EC adder. This work implements a projective Weierstrass EC adder consisting of 128 pipeline cycles. The result is stored at the same memory address when the EC addition is performed.

In contrast, during the bucket aggregation step, the inputs of the EC adder are statically scheduled. The bucket segmentation technique is implemented using a finite-state machine.

After the bucket aggregation step, a single point in projective coordinates is written back to memory using the AXI stream interface. Afterward, the memory interface is informed that the iteration is completed, and the next iteration can start with the bucket accumulation step. The data flow for each iteration is almost identical. However, the *Point Selection* module will discard different point-subscalar pairs, and the bucket aggregation step will differ slightly according to Equation 10.

6 Results

6.1 Memory Requirements

This section compares the memory implications of the suggested approach with recent works. It covers on-chip memory for the buckets, off-chip memory for the points, its potential precomputed values and the corresponding scalar, as well as memory bandwidth requirements for transferring the data onto the FPGA. Table 3 summarizes the theoretical formulas for each work.

[Xav22] uses neither the signed-digit representation nor the BGMW method to reduce the on-chip memory requirements. Therefore, they implemented W sets of 2^c buckets. They implemented a projective Weierstrass EC adder [RCB15] where each EC point consists of three coordinates, with $\mathcal{M}$ the bit-size of one coordinate. They only store two of these coordinates off-chip, as one of the three is redundant. All off-chip stored values must be transferred onto the FPGA, and every point-scalar combination will give rise to W computations.

[IS24] uses the signed-digit representation to reduce the number of buckets by half. Furthermore, they implemented an EC adder that uses the affine Weierstrass representation of Elliptic Curves [GSD11]. Therefore, they only need to store two coordinates for each EC point on-chip and off-chip. As they implemented three EC adders onto the U250 FPGA, their bandwidth requirements are three times higher to fill the pipeline.

[RDQY23] also uses the signed-digit representation. For cheaper EC addition on the BLS12-377 curve, they use the Extended Twisted Edward representation of Elliptic Curves [HWCD08]. This requires four coordinates for each EC point to be stored on-chip, and three coordinates off-chip.

[LFG23] uses the BGMW method to reduce the number of bucket sets to one. On top of that, they use even more off-chip precomputation and the signed-digit representation to be able to reduce the required storage elements to $2^{c-2.25}$. To be able to achieve this, they require $3W$ precomputed values for every EC point, as they compute $sP_i, s2^cP_i, \ldots, s2^{(W-1)c}P_i$ for $s \in \{1, 2, 3\}$. All these precomputed values must also be loaded onto the FPGA.

In this work, the BGMW method is used to reduce on-chip memory and combined with the efficiently computable endomorphism to halve the necessary off-chip memory and bandwidth. The signed-digit representation also saves a factor of two in on-chip memory. Finally, the iterations reduce the on-chip memory by a factor $2^{\log_2(I)}$, but increase the bandwidth requirements by a factor I.

Table 3: Theoretical comparison of required on-chip memory, off-chip memory and memory bandwidth between different works. $\mathcal{M}, b, c$, and W denote respectively the size of one EC point coordinate, the size of one scalar, the window size, and the number of windows.

	On-chip memory	**Off-chip memory**	**Memory bandwidth**
PipeMSM [Xav22]	$3\mathcal{M} \cdot W \cdot (2^c - 1)$	$(2\mathcal{M} + b) \cdot N$	$\frac{2\mathcal{M}+b}{W} \cdot f$
Yrrid Software Ingonyama [IS24]	$2\mathcal{M} \cdot W \cdot 2^{c-1}$	$(2\mathcal{M} + b) \cdot N$	$3 \cdot \frac{2\mathcal{M}+b}{W} \cdot f$
HardCaml [RDQY23]	$4\mathcal{M} \cdot W \cdot 2^{c-1}$	$(3\mathcal{M} + b) \cdot N$	$\frac{3\mathcal{M}+b}{W} \cdot f$
Luo et al. [LFG23]	$2\mathcal{M} \cdot 2^{c-2.25}$	$(3W \cdot 2\mathcal{M} + b) \cdot N$	$\frac{2\mathcal{M} \cdot 3W+b}{W} \cdot f$
This Work	$3\mathcal{M} \cdot 2^{c-1-\log_2(I)}$	$(\frac{W}{2} \cdot 2\mathcal{M} + b) \cdot N$	$I \cdot \frac{2\mathcal{M} \cdot W/2+b}{W} \cdot f$

6.2 FPGA Utilization

Table 4 reports the utilization of FPGA resources and compares them with recent works. It also reports the memory requirements for an MSM of size $N = 2^{24}$. It's worth noting that some works implemented the MSM for the BLS12-377 curve. This curve has slightly smaller coordinates and, more importantly, can use the Extended Twisted Edward coordinate representation [HWCD08]. In this representation, an EC addition only requires eight modular multipliers (or seven if a mixed EC addition is implemented) compared to the twelve modular multipliers in the projective Weierstrass coordinate representation.

Due to the limited number of DSP blocks on the FPGA and their heavy use in modular multiplier units, we chose to use a memory-based modular reduction circuit [WC94] that doesn't rely on any DSP blocks but uses more LUTs instead. As a result, even though we implemented more modular multipliers per compute unit, the DSP block usage per EC adder is lower compared to [RDQY23] and [IS24].

As Luo et al. [LFG23] implemented their work on a CPU, they were not limited by on-chip memory. To compare their work with others, we assume $c = 19$ and $W = 14$ for realistic on-chip memory usage. It's worth noting that even with this large window size, that reduces the total number of windows, their off-chip memory requirements still exceed the typical available quantities. For example, the maximum available off-chip memory on the U55C FPGA is 16 GB. This shows that their additional precomputation optimization can not be used in realistic FPGA implementations of large MSM sizes.

Table 4: Comparison of utilized resources between different works for an MSM size of $N = 2^{24}$.

	LUT	**REG**	**DSP**	**On-chip Memory (MB)**	**Off-chip Memory (GB)**	**Memory bandwidth (bits/cycle)**
[Xav22]†	-	-	-	12.15	1.97	46
[IS24]*	849312	946253	9011	7.44	1.99	153
[RDQY23]†	388000	731000	2999	14.73	1.97	69
[LFG23]*	-	-	-	10.01	63.00	2304
This Work*	721144	885103	2147	8.93	10.92	1597

*EC Curve: BLS12-381; $\mathcal{M} = 381$; $b = 255$
†EC Curve: BLS12-377; $\mathcal{M} = 377$; $b = 255$

6.3 Latency Comparison

Table 5 displays the formulas for the number of EC additions for an MSM of size N for this and recent works. Moreover, for each work it provides the implemented window size and number of windows. The effective number of EC additions for an MSM of size $N = 2^{24}$ is calculated using these data. This figure of merit highlights the advantages of our implemented optimizations. Compared to [Xav22, IS24, RDQY23], we reduced the number of operations by at least 30%. Compared to [LFG23], we find nearly identical results. However, their design requires significantly more off-chip memory, as mentioned in Subsection 6.2, which makes it impossible to implement on FPGA.

Our hardware implementation of the MSM algorithm consists of four iterations. As no data have to be transferred between iterations, it is possible to parallelize these four iterations over multiple compute units on the same device or over multiple devices. The novel distribution with offset technique was used to achieve the most optimal speedup when parallelizing the iterations over multiple compute units or multiple devices. In this work, we implemented and tested the latency improvements using four U55C FPGAs. As can be seen in Table 6, this results in an additional 3.96× (down to 2.47×) speedup compared to the latency results of a single U55C.

Table 6 and Figure 7 compare the latency results of this work with recent designs found in the literature. This comparison includes results from the recent ZPrize competition [ZPr], which focuses on accelerating Zero-Knowledge algorithms, including MSM. Our four compute unit setup achieves a minimal 7.46× (up to 12.77×) speedup compared to the ZPrize 2022 winners [RDQY23]. Also, compared to the ZPrize 2023 winners [IS24], open-sourced in April 2024, we achieved a 2.73× (up to 8.63×) speedup.

Table 6 and Figure 7 also include multiple sizes of the MSM computation to indicate how the different designs scale. Bear in mind that the size of the MSM only affects the size of the bucket accumulation step, while the bucket aggregation step is independent of the number of EC points and scalars. Our design focused on reducing the complexity of the bucket aggregation step. Consequently, the time required for bucket aggregation is small. This enables us to compute over a wide range of MSM sizes without this step becoming dominant.

Table 5: Comparison between recent works of theoretical and applied number of EC additions for an MSM size of $N = 2^{24}$. c, W denote respectively the window size and the number of windows.

	Theoretical number of EC additions	c	W	#ECadd ($\times 10^6$)
[Xav22]	$WN + W \cdot 2 \cdot (2^c - 1) + (W-1)(c+1)$	12	22	369.28
[IS24]	$WN + W \cdot 2 \cdot 2^{c-1} + (W-1)(c+1)$	13	21	352.49
[RDQY23]	$WN + W \cdot 2 \cdot 2^{c-1} + (W-1)(c+1)$	13	20	335.71
[LFG23]	$WN + 2 \cdot 2^{c-2.25}$	19	14	235.10
This Work	$WN + 2 \cdot 2^{c-1} + I + c - \log_2(I)$	19	14	235.41

7 Conclusion

The Multi-Scalar Multiplication algorithm is known to take a significant portion of computation time in Zero-Knowledge Proofs due to its complexity. An optimized version of the Pippenger algorithm was implemented on the U55C FPGA and reduced the computation time of a Multi-Scalar Multiplication of size 2^{24} down to 231 ms using four compute units.

Improvements on top of the Pippenger algorithm were explored and combined. The efficiently computable endomorphism, precomputed multiples of the points, the signed-digit representation, and the novel iteration technique allowed us to push for larger window

Table 6: Comparison of utilized resources between different works for an MSM size of $N = 2^{24}$.

		This Work		[LFG23]	[IS24]*	[RDQY23]†	[Xav22]
FPGA/CPU		U55C		M1 Pro	U250	VU9P	U55C
Frequency		260 MHz		3.2 GHz	250 MHz	278 MHz	125 MHz
#CU		1	4	1	3	1	1
MSM Size	2^{18}	18 ms	7.3 ms	1.49 s	63 ms	-	69 ms
	2^{19}	32 ms	11 ms	2.83 s	76 ms	-	137 ms
	2^{20}	60 ms	18 ms	5.51 s	95 ms	-	273 ms
	2^{21}	117 ms	33 ms	10.7 s	136 ms	-	-
	2^{22}	231 ms	61 ms	-	202 ms	779 ms	-
	2^{23}	459 ms	118 ms	-	344 ms	1 092 ms	-
	2^{24}	914 ms	231 ms	-	631 ms	1 724 ms	-

*ZPrize Winner 2022
†ZPrize Winner 2023

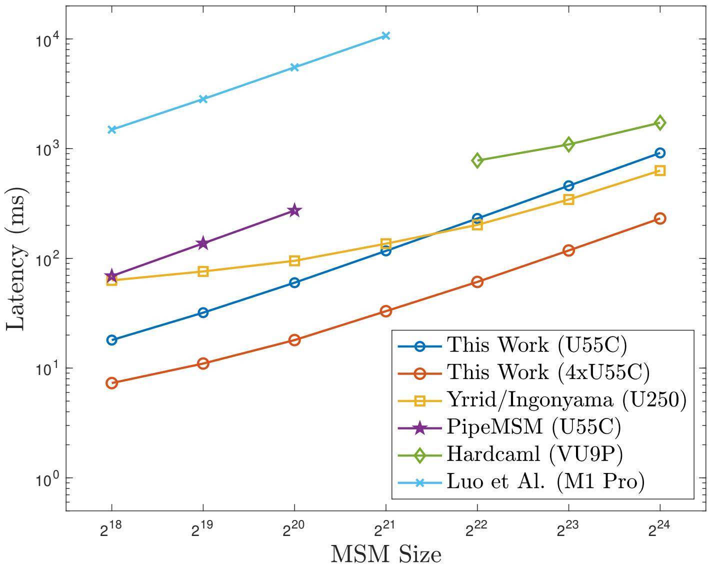

Figure 7: Latency of MSM computation for different MSM sizes

sizes without running out of on-chip memory. As a result, a window size of 19 bits was reached, resulting in fourteen subscalars per scalar, while only implementing 2^{16} buckets.

Moreover, pipeline stalls were limited to 0.7% by simultaneously presenting two potential operations to the EC adder, aiming to find one without collisions during the bucket accumulation step. Additionally, bucket segmentation was implemented to eliminate 98% of the pipeline stalls during the bucket aggregation step.

Furthermore, the novel distribution with offset technique allowed us to optimally distribute the iterations over multiple compute units, ensuring no single iteration became the bottleneck. This resulted in a computation time of 231 ms for an MSM of size 2^{24} using four compute units. This is a ×3.96 speedup compared to our single compute unit case, further validating the effectiveness of the proposed technique.

Although the results of this work are applied to the BLS12-381 EC Curve, all discussed optimizations can be applied to other EC curves as long as the corresponding EC adder is implemented. Moreover, all BLS and BN curves can benefit from the efficiently computable endomorphism to reduce the memory bandwidth requirements by half. EC curves that do not have this property can still use the novel iteration technique and the other implemented optimizations to achieve the same latency results but require double the memory bandwidth.

Acknowledgements This project has received funding, in part, from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement nr. 101020005) and from the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-21-C-0034. Wouter Legiest is funded by FWO (Research Foundation – Flanders) as Strategic Basic (SB) PhD fellow (project number 1S57125N). Michiel Van Beirendonck is funded by FWO as Strategic Basic (SB) PhD fellow (project number 1SD5621N).

erc
European Research Council
Established by the European Commission

References

[ABC+22] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. FPGA acceleration of multi-scalar multiplication: Cyclonemsm. *IACR Cryptol. ePrint Arch.*, page 1396, 2022.

[BDLO12] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. *IACR Cryptol. ePrint Arch.*, page 549, 2012.

[BGMW92] Ernest Brickell, Daniel Gordon, Kevin McCurley, and David Wilson. Fast exponentiation with precomputation (extended abstract). pages 200–207, 01 1992.

[BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. *IACR Cryptol. ePrint Arch.*, page 88, 2002.

[BS64] Richard Bellman and EG Straus. Addition chains of vectors (problem 5125). *The American Mathematical Monthly*, 71(7):806–808, 1964.

[Cha] Chao. Barrett reduction. https://hackmd.io/@chaosma/SyAvcYFxh. (accessed: 02.05.2024).

[Col26] John Colson. A short account of negativo-affirmative arithmetick, by mr. john colson, f. r. s. *Philosophical Transactions (1683-1775)*, 34:161–173, 1726.

[dR94] Peter de Rooij. Efficient exponentiation using procomputation and vector addition chains. volume 950, pages 389–399, 05 1994.

[Gag17] Paul Gagniuc. *Markov Chains: From Theory to Implementation and Experimentation*. 05 2017.

[GLV01] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.

[GSD11] Samta Gajbhiye, Monisha Sharma, and Samir Dashputre. A survey report on elliptic curve cryptography. *International Journal of Electrical and Computer Engineering (IJECE)*, 1, 10 2011.

[Gut20] G. Gutoski. zK hack: Youtube. [Online], 2020. Available: https://www.youtube.com/watch?v=Bl5mQA7UL2I.

[HM09] Mark Hamilton and William P. Marnane. Fpga implementation of an elliptic curve processor using the glv method. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 249–254, 2009.

[HWCD08] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. *IACR Cryptol. ePrint Arch.*, page 522, 2008.

[IS24] Ingonyama and Yrrid Software. Deep dive into the latest msm hardware implementation. https://www.ingonyama.com/blog/deep-dive-into-the-latest-msm-hardware-implementation, 2024. (accessed: 05.05.2024).

[KHSS22] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Zk-img: Attested images via zero-knowledge proofs to fight disinformation, 2022.

[KR17] Tommy Koens and Coen Ramaekers. Efficient zero-knowledge range proofs in ethereum. 2017.

[LFG23] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):358–380, Mar. 2023. Artifact available at https://artifacts.iacr.org/tches/2023/a7.

[LL94] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems (preliminary version). In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 258–263. IEEE Computer Society, 1976.

[PJNB10] Geovandro C. C. F. Pereira, Marcos A. Simplício Jr, Michael Naehrig, and Paulo S. L. M. Barreto. A family of implementation-friendly BN elliptic curves. Cryptology ePrint Archive, Paper 2010/429, 2010.

[RCB15] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. *IACR Cryptol. ePrint Arch.*, page 1060, 2015.

[RDQY23] Andy Ray, Ben Devlin, Fu Yong Quah, and Rahul Yesantharao. Hardcaml zprize submission. https://zprize.hardcaml.com, 2023. (accessed: 01.02.2024).

[Sch] Peter Schwabe. Scalar-multiplication algorithms. https://cryptojedi.org/peter/data/eccss-20130911b.pdf. (accessed: 01.02.2024).

[WC94] Cheng-Wen Wu and Yung-Fa Chou. General modular multiplication by block multiplication and table lookup. In *1994 IEEE International Symposium on Circuits and Systems, ISCAS 1994, London, England, UK, May 30 - June 2, 1994*, pages 295–298. IEEE, 1994.

[Xav22] Charles F. Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. *IACR Cryptol. ePrint Arch.*, page 999, 2022.

[Xila] Xilinx. Alveo data center accelerator card platforms user guide (ug1120). https://docs.amd.com/r/en-US/ug1120-alveo-platforms/U55C. (accessed: 01.02.2024).

[Xilb] Xilinx. Vivado design suite: Axi reference guide (ug1037). https://docs.amd.com/v/u/en-US/ug1037-vivado-axi-reference-guide. (accessed: 01.02.2024).

[ZPr] ZPrize. Accelerating zero-knowledge proofs competition. https://www.zprize.io. (accessed: 05.05.2024).