FPGA Acceleration of Multi-Scalar Multiplication: CycloneMSM

Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder and Javier Varela

Jump Trading | Jump Crypto
{kaasaraai,jvarela}@jumptrading.com
{dbeaver,ecesena,rmaganti,nicolas}@jumpcrypto.com

October 14, 2022

Abstract. Multi-Scalar Multiplication (MSM) on elliptic curves is one of the primitives and bottlenecks at the core of many zero-knowledge proof systems. Speeding up MSM typically results in faster proof generation, which in turn makes ZK-based applications practical.
We focus on accelerating large MSM on FPGA, and we present speed records for `BLS12-377` on FPGA: 5.66s for $N = 2^{26}$, sub-second for $N = 2^{22}$.
We developed a fully pipelined curve adder in extended Twisted Edwards coordinates that runs at 250MHz. Our architecture incorporates a scheduler to reorder curve operations, that's suitable not just for hardware acceleration, but also for software implementations using affine coordinates with batch inversion. The software implementation achieves $+10 - 20\%$ performance improvement over the state-of-the-art `gnark-crypto` library.

Keywords: Multi-Scalar Multiplication, Elliptic Curve Cryptography, FPGA Design, Zero-Knowledge Proofs

1 Introduction

Multi-Scalar Multiplication (MSM) on an elliptic curve refers to the computation of a linear combination of points in the group. Given N points P_i and scalars n_i, compute:

$$R = n_1 P_1 + n_2 P_2 + \cdots n_N P_N = \sum_{i=1}^{N} n_i P_i \ .$$

The special case for $N = 2$ is key to many real-world applications, like verification of ECDSA signatures, or speeding up single scalar multiplication via endomorphisms, when available.

In recent years, the interest for larger N has grown in relationship to a variety of use cases, including batch signature verification, homomorphic encryption, and Zero-Knowledge (ZK) proofs.

Perhaps surprisingly, when dealing with large N such as $N \geq 2^{20}$, we know how to compute a MSM in just $13 - 17N$ group operations with the bucket method, attributed to [Pip76], outperforming the naive approach to compute N individual scalar multiplication and add the results.

While the algorithm is relatively well understood, practical implementations on modern platforms involving the optimization and tuning of the various parameters still constitute

an open area of research. Efficient software implementations of MSM can be found in open source libraries, notably `arkworks`[1] and `gnark-crypto`[2].

In this work we analyze the problem of computing MSM on FPGA devices, for size of N that are significantly larger than on-chip memory, e.g. $N = 2^{26}$. We show that the techniques that we develop for hardware also lead to efficient software implementations, improving upon the current state-of-the-art.

Our work is motivated by ZK applications, particularly in the context of blockchain technology. ZK proofs are a building block to build both scalability solutions (e.g., ZK-rollups) and privacy-enhanced applications for blockchains. In many concrete schemes, generating a ZK proof requires computing a MSM, typically on a fairly large dataset.

Contribution

We present CycloneMSM, an implementation of Multi-Scalar Multiplication (MSM) on `BLS12-377`, accelerated via FPGA. The implementation is *optimized* to compute a fixed-base MSM on `BLS12-377` for $N = 2^{26}$ independent uniformly random (UR) scalars, with lowest latency. The implementation is correct for any set of scalars.

Architecture (Section 3). Our architecture has 3 main components: 1) a curve adder, 2) a MSM controller, and 3) a scheduler to reorder operations and maximize performance. We analyze the real-world case of UR scalars and present efficient schedulers for hardware acceleration, as well as a software implementation using affine coordinates with batch inversion.

FPGA Design (Section 4). We implement the bucket algorithm with a window size of $c = 16$ using extended Twisted Edwards coordinates, available for `BLS12-377` and `Bandersnatch`. To accumulate points into buckets we use `MixedAdd` which only requires 7 `mul` in the field, thus reducing the circuit size.

Our FPGA runs at 250MHz, i.e., it can begin a new curve addition every 4ns. The curve adder is *fully pipelined* and has length 96 stages. The controller implements a PCIe interface between host and FPGA, a DDR (external memory) from/to SRAM (on-chip memory) interface, and logic to accumulate points into buckets and aggregate buckets. The scheduler runs on the FPGA, detects conflicting points (points with the same reduced scalar) and stores them into a FIFO backed by SRAM and DDR, thus delaying their processing.

Performance (Section 5). The design uses 525k LUTs, 661k registers, 404 BRAMs, 219 URAMs, 2277 DSPs and 3 DDR channels.

On average, our implementation computes a MSM for $N = 2^{26}$ UR scalars in 5.66s, consuming 51W. For comparison, a purely software implementation in `gnark-crypto` takes about 24s on an 18-core Intel Core i9-7980XE CPU at 4.8GHz, which consumes 530W.

For completeness, we show that the software implementation based on `gnark-crypto`, using affine coordinates and batch inversion, achieves +10-20% performance improvement over the original library.

Related Works

The bucket algorithm to compute MSM is described in [BDLO12, Gut20] and attributed to [Pip76]. While the asymptotic complexity and the high-level algorithm are understood, most work in literature focuses on relatively small input sizes (typically $N \leq 2^{20}$). Larger values of N require streaming data to/from external memory devices and scheduling point additions out-of-order to maximize performance.

[1] https://github.com/arkworks-rs
[2] https://github.com/ConsenSys/gnark-crypto

Our work is motivated by participation in the ZPrize[3], a competition among industry partners and academic institutions to accelerate primitives related to zero-knowledge cryptography. In this context, a similar work to ours is [Xav22]. At the time of writing, the authors did not release information for large N, e.g. $N = 2^{26}$, and linearly interpolating our work appears to be over 40% faster.

Broadening the view on ZK applications, PipeZK [ZWZ+21] introduces a framework for hardware acceleration of various primitives, including MSM. We share their long term view and consider our work one of the key building blocks to achieve high-performance ZK applications through FPGA acceleration.

Several works in literature focus on improving the algorithmic efficiency of MSM. [SIM12] focus on using redundant bit representations to reduce the number of bit operations. [DKS09, OS02, SZZG21, OS03, LTD08, SS14] take advantage of redundant bit representations (or double base bit representations) with lower Hamming weights to reduce the complexity of multipliers in hardware. Other approaches [MSZ21] take advantage of the structure of the curve to create efficient precomputation strategies, including endomorphisms. A more application-specific implementation of MSM is detailed here: [BCG+18]. [DDQ07] also presents a good survey of efficient hardware implementations of elliptic curve operations.

While our work targets specifically `BLS12-377` due to the ZPrize specifications, many other curves are growing in popularity, see e.g. [AEHG22] for a survey. Noteworthy is `Bandersnatch` [MSZ21]. Generally speaking our FPGA design is adaptable to changing base field and curve, granted we're using extended Twisted Edwards coordinates.

Finally, we want to note that large MSM instances are explored in [BCHO22] where the authors report that a MSM for $N = 2^{35}$ took several days to compute and they "stopped at these sizes only due to time constraints". Although we didn't implement these large sizes yet, we expect to bring the computation time down to a few hours, thus paving the path for a completely new set of ZK schemes and applications that requires proof of very large circuits, such as zero-knowledge virtual machines.

2 Background

2.1 Elliptic Curves, Twisted Edwards

Let $\mathbb{F}_q$ be a finite field of prime order $q > 3$. An elliptic curve E over $\mathbb{F}_q$ is a non-singular curve defined by the Weierstrass equation:

$$y^2 = x^3 + ax + b \ , \qquad \text{with } a, b \in \mathbb{F}_q \ .$$

For cryptographic applications we focus on curves whose group of rational points $E(\mathbb{F}_q)$ has order divisible by a large prime r. We denote by $\mathbb{G} \subseteq E(\mathbb{F}_q)$ the subgroup of order r, and $P_\infty \in \mathbb{G}$ the point at infinity.

Let E^T be a Twisted Edwards curve given by the equation[4]:

$$-x^2 + y^2 = 1 + \frac{k}{2}x^2y^2 \ , \qquad \text{with } k \in \mathbb{F}_q \ . \tag{1}$$

Every Twisted Edwards curve is birationally equivalent to an elliptic curve in Weierstrass form. The converse only holds for some curves.

Example (`BLS12-377` [BCG+18])**.** Let q be the 377-bit prime shown below. The elliptic curve $E\colon y^2 = x^3 + 1$ over $\mathbb{F}_q$ is called `BLS12-377`. This is a pairing-friendly curve with embedding degree 12. The group $E(\mathbb{F}_q)$ has order rc, where r is a 253-bit prime. `BLS12-377`

[3] https://zprize.io

[4] A more general definition uses $ax^2 + y^2 = 1 + dx^2y^2$, here we restrict to the case $a = -1$.

admits a Twisted Edwards representation E^T as in (1), with k given below. Throughout this paper, $\mathbb{G}$ will denote the subgroup of order r of either $E(\mathbb{F}_q)$ or $E^T(\mathbb{F}_q)$, depending on the context.

$q =$ 0x01ae3a4617c510eac63b05c06ca1493b1a22d9f300f5138f1ef3622fba094800170b5d44300000008508c00000000001

$r =$ 0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000001

$c =$ 170b5d44300000000000000000000000

$k =$ 0x0196bab03169a4f2ca0b7670ae65fc7437786998c1a32d217f165b2fe0b32139735d947870e3d3e4e02c125684d6e016

Several coordinate systems have been developed to improve efficiency of operations in $\mathbb{G}$ under different assumptions. We refer to the Explicit-Formulas Database [BL07] for a complete list. For the purpose of evaluating performance, we denote by `Dbl`, `Add` resp. the double and add operations in $\mathbb{G}$, and by `mul`, `inv` resp. multiplication[5] and inversion in the field $\mathbb{F}_q$ (capitalized are curve operations, lowercase are field operations).

In this work we use the following coordinate systems. For Weierstrass curves: affine coordinates (x, y), and extended Jacobian coordinates $(X : Y : Z : W)$, where $x = X/Z$, $y = Y/W$, $Z^3 = W^2$. We denote `MixedAdd` the addition where the second operand is in affine coordinates, or equivalently $Z = W = 1$. For Twisted Edwards curves: extended affine coordinates (x, y, u) with $u = kxy$ (note that we included k), and extended projective $(X : Y : Z : T)$, where $x = X/Z$, $y = Y/Z$, $T = XY/Z$. Analogously, we denote `MixedAdd` the addition where the second operand is in extended affine coordinates.

In Table 1 we provide a summary of the coordinate systems used in this work, comparing with the state-of-the-art implementation in `gnark-crypto`.

Table 1: Coordinate systems used in this work.

Hardware	Software	
Twisted Edwards This work	Weierstrass This work	`gnark-crypto`

Extended Projective
`MixedAdd`: 7 `mul`
`Add`: 9 `mul`
(unified addition)
	Affine
	`Add`: 2 `mul` + 1 `inv`
	`Dbl`: 3 `mul` + 1 `inv`
	and extended Jacobian
		Extended Jacobian
		`MixedAdd`: 10 `mul`
		`Add`: 14 `mul`
		`Dbl`: 10 `mul`

2.2 Bucket Algorithm

Our goal is to compute a multi-scalar multiplication (MSM) for large N, e.g. $N = 2^{26}$. Given points $P_i \in \mathbb{G}$ and scalars $n_i \in \mathbb{Z}_r$, $i \in [0..N)$, we want to compute:

$$R = \sum_{i=0}^{N-1} n_i P_i \in \mathbb{G} \ .$$

First, we're going to describe an efficient algorithm to compute a *reduced* MSM:

$$\sum_{i=0}^{N-1} \bar{n}_i P_i = \bar{R} \ ,$$

where $\bar{n}_i < 2^c$ for a small constant c, for example $c = 16$. Next, we'll use this algorithm to compute the full MSM.

[5] For simplicity we won't distinguish between multiplication and square in the field.

Let *bucket* $\mathcal{B}_k$, $k \in [0..2^c)$ be the set of points whose reduced scalar is equal to k: $\mathcal{B}_k = \{P_i \mid \bar{n}_i = k\}$, and S_k the sum of all points in $\mathcal{B}_k$:

$$S_k = \sum_{P \in \mathcal{B}_k} P \tag{2}$$

The MSM can be rewritten as:

$$\sum_{i=0}^{N-1} n_i P_i = \sum_{k=0}^{2^c-1} k S_k = \sum_{k=1}^{2^c-1} k S_k \ .$$

An efficient algorithm to compute the reduced MSM works as follow:

1. Bucket accumulation phase: add each point P_i to the corresponding bucket sum S_k.
2. Bucket aggregation phase: efficiently compute $\sum_{k=1}^{2^c-1} k S_k$, via:

$$\sum_{k=1}^{T} k S_k = S_T + (S_T + S_{T-1}) + \cdots + (S_T + S_{T-1} + \cdots + S_1) \ .$$

Let's now return to the full MSM. The idea is to split the scalars in W windows of c bits each, $W = \lceil \frac{1}{c} \log r \rceil$, compute the W reduced MSM and aggregate the final result. For every $j \in [0..W)$, denote: $n_i^{(j)} = n_i / 2^{jc} \mod 2^c$. Then:

$$R^{(j)} = \sum_{i=0}^{N-1} n_i^{(j)} P_i$$

is a reduced MSM that can be computed with the algorithm above, and finally:

$$R = \sum_{j=0}^{B-1} 2^{jc} R^{(j)} \ .$$

Algorithm 1 summarizes the whole process, including a few well-known optimizations:

- Use mixed additions for faster bucket accumulation.
- Use signed scalars, thus 2^{c-1} of buckets instead of 2^c, which leads to less memory and most importantly half time for bucket aggregation.

Finally, let's review the computational cost, with particular focus on the case where E is `BLS12-377`, $\log r = 253$-bit scalars, $N = 2^{26}$ points, $c = 16$-bit reduced scalars, $W = 16$ windows. For each window, i.e. W times:

- Bucket accumulation[6]: N `MixedAdd`
- Bucket aggregation: 2^c `Add`
- Final result aggregation: c `Dbl` + 1 `Add` (negligible)

For sufficiently large N the computation is $O(N)$, dominated by the WN `MixedAdd`.

[6] One could exclude adding points with reduced scalar = 0 (on average $N/2^{c-1}$ for UR scalars), and adding points for the first time in each bucket (2^{c-1} buckets). This leads to $N - N/2^{c-1} - 2^{c-1}$ `MixedAdd`. For large N, this is approximately N `MixedAdd`.

Algorithm 1 Bucket Method for Multi-Scalar Multiplication

```
1:  function MSM_INIT(points)
2:      convert points, e.g. to Edwards
3:      initialize context ctx
4:      return ctx

5:  function MSM(ctx, scalars)
6:      PREPROCESS(scalars)
7:      R ← P_∞
8:      for j = W − 1 downto 0 do
9:          ACCUMULATE(j, ctx, scalars)
10:         R^(j) ← AGGREGATE(j, ctx)
11:         R ← R + R^(j)
12:         if j > 0 then R ← 2^c R
13:     return R

14: function ACCUMULATE(j, ctx, scalars)
15:     reset buckets
16:     for i = 0 to N − 1 do
17:         (k, sgn) ← REDUCE(scalars[i], j)
18:         P ← sgn · ctx.points[i]
19:         S_k ← S_k + P                ▷ MixedAdd

20: function AGGREGATE(j, ctx)
21:     K ← 2^(c−1) − 1
22:     R̄ ← P_∞      T ← S_K
23:     for k = K − 1 downto 1 do
24:         R̄ ← R̄ + T                   ▷ Add
25:         T ← T + S_k
26:     return R̄
```

3 CycloneMSM

Equation (2), i.e. bucket accumulation, hides a detail that is critical for efficient implementations: the order in which points are accumulated into buckets. Most existing software implementations (e.g., `arkworks`, `gnark-crypto`) process points in the order they were given.

In this section we generalize Algorithm 1 by introducing a scheduler that controls the order in which points are accumulated into buckets, we derive a scheduler that optimizes bucket accumulation assuming UR scalars, and we present two relevant examples: FPGA acceleration, and the use of affine coordinates with batch inversion.

3.1 Architecture

We present our architecture in Figure 1, composed of three main components: Curve Adder, MSM Controller and Scheduler.

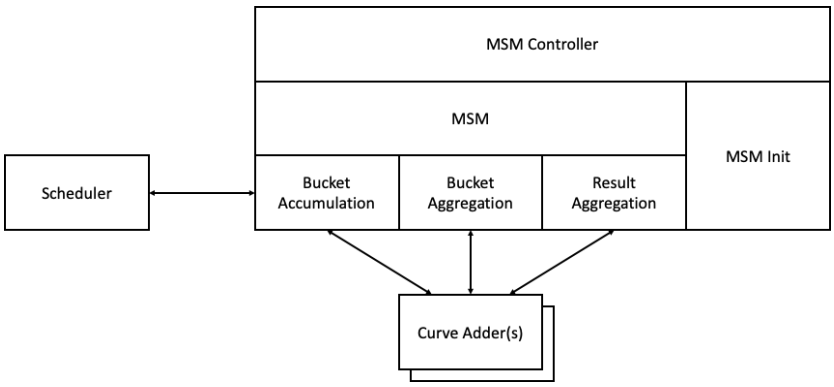

Figure 1: System architecture comprised of Curve Adder, MSM Controller and Scheduler.

Curve Adder. One or multiple elliptic curve addition operators. Multiple adders may be used in different phases of the bucket algorithm, for example to compute in different coordinate systems.

MSM Controller. Responsible to handle the workflow of the MSM.

MSM Init. Initialize the system, store points, optionally perform pre-computaiton on points, e.g. conversion to Twisted Edwards coordinates.

MSM. Compute the actual MSM, given N scalars. We implement the bucket algorithm as W iterations of a bucket accumulation, then bucket aggregation, and final result aggregation phases. Each phase relies on a Curve Adder for actual computation. Bucket accumulation is driven by a Scheduler to maximize performance.

Scheduler. Responsible for optimizing the order of operations during bucket aggregation, to maximize performance. We will describe scheduling in detail in the next section.

Our main goal is to accelerate MSM computation via FPGA. However, for large N, the same architecture can be reused for software implementations, e.g. to speed up the computation using affine coordinates and batch inversion. In the next sections we are going to describe in details the benefits of scheduling and these two applications.

3.2 Scheduler for UR Scalars

At a high level, during the bucket accumulation phase, we want to enforce that T consecutive point additions are independent.

We model the scheduler as an iterator that, at discrete time $t = 0, 1, 2, \ldots$, returns a reduced scalar $k^{(t)}$ and point $P^{(t)}$. We want to compute $S_{k^{(t)}} \leftarrow S_{k^{(t)}} + P^{(t)}$ but we want to enforce the condition that:

$$k^{(t_1)} \neq k^{(t_2)} \qquad \text{if } |t_1 - t_2| \leq T \ . \tag{3}$$

The idea is that, when all T buckets different, the T additions can be performed more efficiently. Or vice versa, if any two buckets are the same, then we need to introduce an artificial delay that makes the MSM less efficient.

We now want to analyze this problem and show that when scalars are UR it is possible to build efficient schedulers that: 1) for correctness, satisfy the condition (3); and 2) for efficiency, only reorder a small percentage points and scalars.

If scalars are UR, so are the reduced scalars for every window. Without loss of generality, let $\{\bar{n}_i\}_{i=0}^{N-1}$, $\bar{n}_i \leq 2^{c-1}$, bet a sequence of UR reduced scalars. Since N is large, we can approximate it as a sequence of discrete random variables following a Poisson distribution, where the rate is $\lambda = N/2^{c-1}$.

If $\mathcal{E}$ is the event of finding a collision: $\bar{n}_j = \bar{n}_i$, with $i < j \leq i+T$. Then, the probability of a collision, $P(\mathcal{E}) \approx NT/2^{c-1}$. For example, for $N = 2^{26}$, $c = 16$ and $T = 100$, we expect to find about $204,800$ collisions, which is about 0.3% of N.

This suggests that by rescheduling a small percentage of the N points and reduced scalars (for each window), we can satisfy condition (3). Let's introduce two examples of schedulers: the delayed and the greedy scheduler.

Delayed scheduler. The idea is simply to delay conflicting points, reprocess them in a second pass, delay again conflicting points to process them in a third pass, etc. It's easy to see that the probability to find collisions in subsequent passes decreases significantly. In practice, for $N = 2^{26}$, rarely 4 passes are needed. The drawback of this approach is that we need to store $NT/2^{c-1}$ points and reduced scalars to recompute them later.

Greedy scheduler. Another approach is to reprocess a point as soon as possible. For example, say $k_i = k_{i+1}$ is a conflict, and k_i is output at time t, i.e. $k_i = k^{(t)}$. Then k_{i+1}, P_{i+1} can be processed at time $t + T + 1$. The advantage of the greedy scheduler is that it needs to maintain a smaller queue of delayed points, since points are dequeued frequently. In fact, the expected maximum length of the queue for $N = 2^{26}$ is about 10, so the greedy scheduler requires very little storage compared to the delayed scheduler.

3.3 Applications: FPGA, Batch Affine

We consider two concrete applications that benefit from reordering operations via a scheduler: hardware acceleration via FPGA, and software implementation in affine coordinates using batch inversion.

In hardware, let's assume that a processor computes a point addition in T stages, i.e. given inputs S_k, P at time t, it produces the output $S_k + P$ at time $t + T$. Then we can pipeline one point addition per clock cycle, provided that any new bucket does not conflict with buckets already inside the pipeline. This is equivalent to condition (3).

In software, the use of affine coordinates for bucket accumulation can provide optimal performance. Addition formulas in affine coordinates require 2 or 3 `mul` + 1 `inv`. By computing additions in batches of T points and by using batch inversion (T `inv` $\rightarrow$ $3T$ `mul` + 1 `inv`), the cost can be reduced down to less than $6T$ `mul` + 1 `inv`. For large enough T this is widely considered the best coordinate system, however none of the major libraries implement MSM in affine coordinates. In order to process additions in batch, the T input buckets need to be different, as expressed by condition (3).

In both cases, the use of a scheduler to reorder the operations can help maximize the throughput of point additions. Said in another way, attempting to process points in order would require to introduce empty bubbles in the FPGA pipeline, or shorten the batch size in affine coordinates, leading to sub-optimal performance.

We stress that cost of scheduling points should be negligible compared to the cost of a single point addition. The schedulers presented in the previous section are designed to perform well under the assumption of UR scalars. It is trivial to build counter-examples for which these schedulers perform poorly, for example the edge case where all scalars are the same. For real-world applications, however, this is not particularly interesting.

4 FPGA Design

4.1 Field Arithmetic

Arithmetic in $\mathbb{F}_q$ is implemented using 377-bit integers in Montgomery representation, using Montgomery parameter $R = 2^{384}$ as in most software implementations.

Because of our choice of coordinates, we only need additions, subtractions and multiplications (`mul`) in the field, and no costly inversion. For `mul` we experimented with a variety of possible implementations, either based on CIOS (the most widely adopted in software) or combinations of schoolbook, Karatsuba and Toom-Cook methods followed by Montgomery reduction. Here we limit the explanation to our final implementation.

Our final field `mul` is built with three 384-bit integer multiplications. The first and last are done with a 3-layer Karatsuba starting from a base 48-bit integer multiplier built with 6 DSPs, as shown in Figure 2. The intermediate one is of the form $ab \bmod R$, with b constant, and where we only need the lower half of the result ($\bmod R$). This is implemented as a custom multiplier based on the NAF representation of b: we aggregate bits of a in a positive and a negative adder, and compute the final result by subtracting of the two.

Finally, note that $R \geq 4q$. It is known that `mul` can accept inputs in redundant representation $a, b \in [0..2q)$, instead of just $[0..q)$. We then use this fact to implement the curve adder and save some modular reductions.

4.2 Constant Multiplier

Montgomery multiplication requires 2 constant multiplications. We use a mix of DSPs and constant multipliers to compute the Montgomery multiplications. To further improve the efficiency of the constant multiplier, we make use of other bit representations, including NAF. A constant multiplier for ab, where a and b are represented as bits, accumulates the

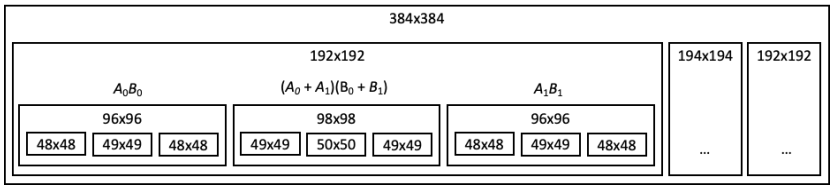

Figure 2: Integer multiplier, used to build field `mul`.

result of shifting a by i for every b_i equal to 1. NAF is a unique signed bit representation that reduces the hamming weight of a bit string s from $\frac{1}{2}$ to $\frac{1}{3}$. NAF effectively splits the adder tree into two components (one corresponding to the indices of the constant b that are positive and one corresponding to the indices of b that are negative). The effect of this is that the depth of the adder tree is reduced from $\log_2 W(b)$ to $\log_3 W(b)$, where $W(b)$ is the hamming weight of b. The adder tree for a constant multiplier with NAF is summarized in Algorithm 2 and shown in Figure 3.

Algorithm 2 NAF Multiplier

function MULTIPLYNAF(a, b)
  // b is constant
  $(b_0, \ldots, b_i \ldots, b_n) \leftarrow \text{bits}(b)$
  $\hat{b} \leftarrow NAF(b)$
  // keep track of indices with 1 and −1
  $acc_{pos} \leftarrow 0$, $acc_{neg} \leftarrow 0$
  for $i = 0$ up to n **do**
    // multiplier is hard-coded b/c $\hat{b}$ is constant
    if $\hat{b}_i$ is 1 **then**
      $acc_{pos} \leftarrow (acc_{pos} + a \ll i)$
    if $\hat{b}_i$ is -1 **then**
      $acc_{neg} \leftarrow acc_{neg} + (a \ll i)$
  $acc \leftarrow acc_{pos} - acc_{neg}$
  return acc

4.3 Curve Arithmetic

We implemented a *fully pipelined* Twisted Edwards adder in extended projective coordinates that computes `MixedAdd`, and can be *reconfigured* to compute a full `Add` or `Dbl` in 2 cycles[7]. The pipeline is shown in Figure 4, runs at 250MHz and has length 96 stages[8], so it can process a new `MixedAdd` every 4ns, returning the result after $96 \times 4\text{ns} = 384\text{ns}$.

Inputs are points $P_1 = (X_1 : Y_1 : Z_1 : T_1)$ in extended projective and $P_2 = (x_2, y_2, u_2)$ in extended affine coordinates, with $u_2 = kx_2y_2$ (i.e. $u_2 = kt_2$, not t_2). Output is $P_1 + P_2$ in extended projective $(X : Y : Z : T)$. We adapted formulas from [HWCD08, Sec. 4.2] to compute a `MixedAdd` in 7 `mul` and also used the 4-processor version of the formulas, as presented in Figure 5, to reduce the pipeline length.

Recall that our field multiplier accepts inputs in $[0..2q)$. Hence we removed modular reductions from additions (all but $2Z_1$) and subtractions. Specifically, if $a, b \in [0..q)$, then $a + b$ is already in $[0..2q)$, and $q + a - b$ is always in $[0..2q)$. This saves a few resources.

A full `Add` requires 9 `mul`, and we implemented it as a 2-cycle pipeline in order to maintain low resource utilization. Denoting $P_2 = (X_2 : Y_2 : Z_2 : T_2)$, at stage 0 we feed

[7] We're using unified formulas, so computing `Dbl` or `Add` is the same.
[8] The length depends on the frequency, for example at 125MHz the length is 65 stages.

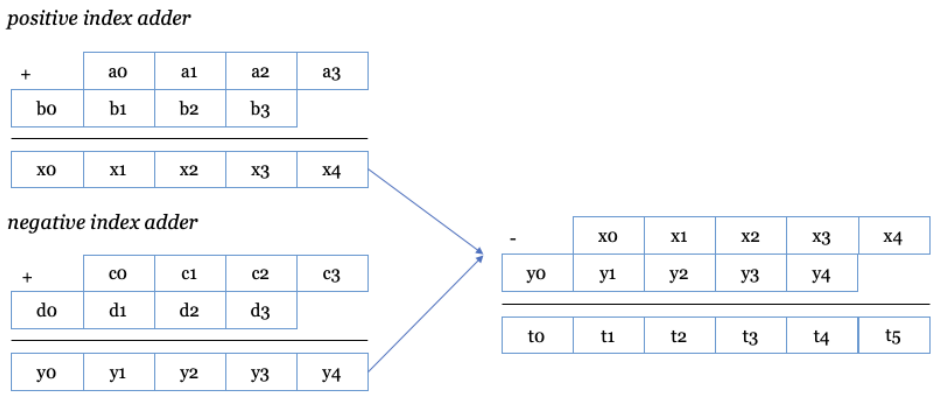

Figure 3: Adder tree with NAF

Z_1, Z_2 into the multiplier, at stage 1 we feed T_2, k. After 39 stages (the length of a single Montgomery multiplier) we retrieve the results $r_0 = Z_1 Z_2$ and $r_1 = kT_2$, and feed `MixedAdd` with inputs: $(X_1 : Y_1 : r_0 : T_1)$ and (X_2, Y_2, r_1).

4.4 MSM Acceleration

We implemented the bucket method assuming fixed-base MSM, and optimized it to achieve low latency for $N = 2^{26}$, in particular trying to achieve linear latency for large enough N.

Figure 6 depicts the FPGA architecture, while the steps are specified in Algorithm 3.

Algorithm 3 CycloneMSM on FPGA

function MSM_INIT(points)
  convert points *to Edwards*
  initialize FPGA + *send* points
  return ctx

function MSM(ctx, scalars)
  PREPROCESS(scalars)
  $R \leftarrow P_\infty$
  for $j = 0$ to $W - 1$ **do**
    ACCUMULATE(j, ctx, scalars)
    $R^{(j)} \leftarrow$ AGGREGATE(j, ctx)
    $R \leftarrow R + R^{(j)}$
    if $j > 0$ **then** $R \leftarrow 2^c R$
  return R

function ACCUMULATE(j, ctx, scalars)
  invoke FPGA.ACCUMULATE(j)
  send reduced scalars

function AGGREGATE(j, ctx)
  $\bar{R} \leftarrow$ FPGA.AGGREGATE(j)
  return $\bar{R}$

function FPGA.ACCUMULATE(j)
  reset buckets
  for (k, P) in SCHED(j, scalars) **do**
    $S_k \leftarrow S_k + P$

function FPGA.AGGREGATE(j)
  $K \leftarrow 2^{c-1} - 1$
  $\bar{R} \leftarrow P_\infty \qquad T \leftarrow S_K$
  for $k = K - 1$ downto 1 **do**
    $\bar{R} \leftarrow \bar{R} + T$
    $T \leftarrow T + S_k$
  return $\bar{R}$

MSM Init. We assumed an initialization phase where inputs are N points in Weierstrass affine coordinates. Points are converted into affine Twisted Edwards $(x, y, u = kxy)$,

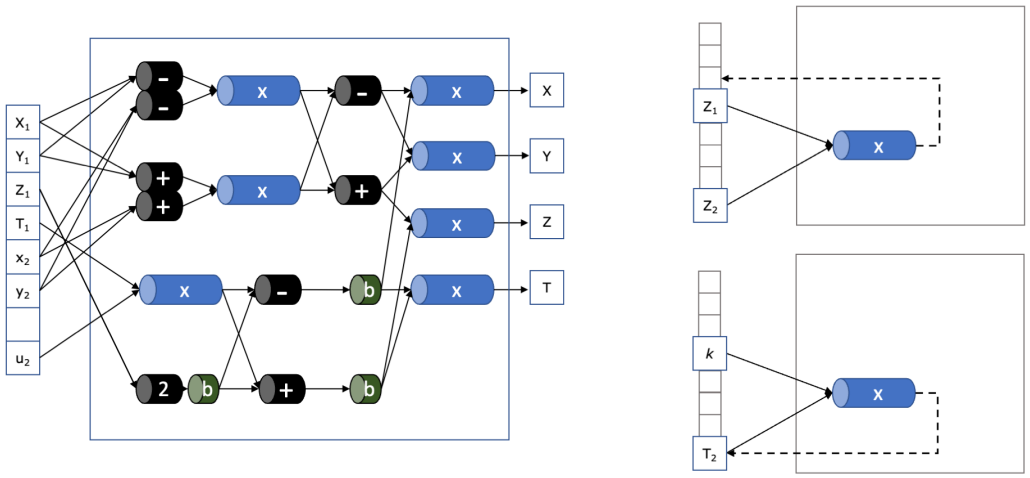

Figure 4: Fully pipelined `MixedAdd` in extended Twisted Edwards coordinates (left), where -, +, 2, × and `b` are resp. subtractions, additions, double, field multiplication and alignment buffers. The pipeline can be reconfigured to just run a field multiplication (right), leading to a multi-cycle pipeline to compute a full `Add`.

Cost	Step	Processor 1	Processor 2	Processor 3	Processor 4
	1	$R_1 \leftarrow Y_1 - X_1$	$R_2 \leftarrow y_2 - x_2$	$R_3 \leftarrow Y_1 + X_1$	$R_4 \leftarrow y_2 + x_2$
3 `mul`	2	$R_5 \leftarrow R_1 R_2$	$R_6 \leftarrow R_3 R_4$	$R_7 \leftarrow T_1 u_2$	$R_8 \leftarrow 2Z_1$
	3	$R_1 \leftarrow R_6 - R_5$	$R_2 \leftarrow R_8 - R_7$	$R_3 \leftarrow R_8 + R_7$	$R_4 \leftarrow R_6 + R_5$
4 `mul`	4	$X \leftarrow R_1 R_2$	$Y \leftarrow R_3 R_4$	$Z \leftarrow R_2 R_3$	$T \leftarrow R_1 R_4$

Figure 5: `MixedAdd` in extended Twisted Edwards coordinates, cf. [HWCD08, Sec. 4.2]

sent to the FPGA and stored in DDR. Conversion from Weierstrass to Twisted Edwards costs approximately like a curve operation, and can be sped up using batch inversion.

MSM. In the remainder of this section we focus on the actual computation of MSM given N 256-bit scalars.

Our FPGA core can compute one `MixedAdd` per clock cycle, i.e. at 250MHz we need to feed one reduced scalar every 4ns. Because of the PCIe bandwidth between host and FPGA, we can receive at most 64-bit/cycle, therefore sending the whole scalars would be sub-optimal.

In the current implementation we send one reduced scalar (16-bit) in every command. Commands are 64-bit long and are sent in batches of 8. We did experiment with various protocols between host and FPGA, but in the final iteration commands fundamentally only contain the reduced scalars. Scalars are sent in order, and the FPGA implements a *delayed scheduler* (cf. Section 3.2) to process points and scalars out of order.

We use signed scalars to reduce the total number of buckets in half. With $c = 16$-bit *signed* reduced scalars we need 2^{15} buckets on the FPGA, i.e. approximately 6MB of SRAM. While the SRAM is not an issue per se, these SRAM blocks need to be wired to the curve adder pipeline. Doubling the number of buckets did not work in our experiments as we could not synthesize a pipeline at 250MHz due to wiring congestion.

Choice of window size c. The choice of $c = 16$ is primarily a trade off between the amount of buckets and the speed of computing the reduced scalars. We estimated that with $c = 17$ the whole algorithm would only be 7% faster, but it would require double the number of buckets and more computation on the host side to reduce the scalars.

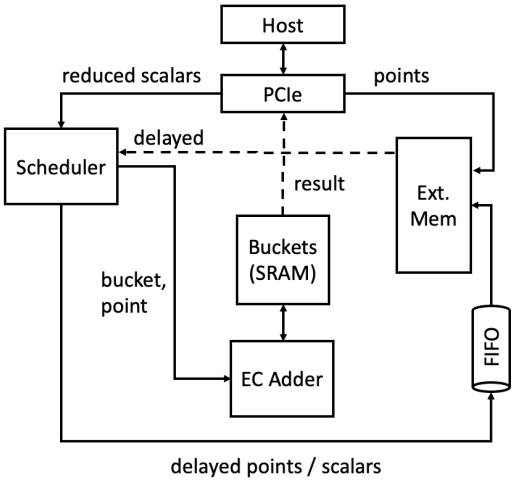

Figure 6: Architecture of FPGA implementation

Still, with $c = 16$ computing signed reduced scalars on the host proved to be challenging. The computation requires to propagate a carry across 16-bit windows. Precomputing $N = 2^{26}$ reduced scalars for all 16 columns takes a few seconds, which is unacceptably high[9]. Computing the reduced scalars on the fly gets slower for higher columns, again making it ineffective. Our latest implementation precomputes 64-bit signed windows (i.e. it propagates carry at 64-bit level), and computes reduced scalars on the fly within these windows, maximizing the speed at which the host can stream reduced scalars to the FPGA. The precomputation is done on a parallel thread, while the host streams the first windows to the FPGA. Precomputing over 64-bit windows and parallelizing this precomputation provided over 15% speed up, from about 7s down to the final 5.66s for $N = 2^{26}$.

Bucket accumulation. As the FPGA receives the reduced scalars, it performs the bucket accumulation phase, i.e., if the i-th reduced scalar is k, the FPGA computes $S_k \leftarrow S_k \pm P_i$. Points are streamed from DDR into SRAM using 3 channels, one per coordinate. Based on the length of the curve adder pipeline T, the FPGA needs to account for *conflicts*. A bucket can only be used once within the T stages of the pipeline (we need to receive the output of the sum before we can use it again as an input). Conflicting points are stored in a FIFO backed by SRAM and DDR. These points are sent to DDR, later re-read and processed in a second pass. Conflicting points at the second pass are stored in the FIFO again and processed at a 3rd pass. Practically speaking, given the assumption of UR scalars, it's very rare to ever get to a 4th pass and the total length of the FIFO after all the passes is about 235,000, so this scheduler is pretty effective even if naive at first sight. We note that the result is always correct, even in the worst case where all the scalars are the same, however performance degrades significantly (roughly speaking 100×, since the pipeline length is about 100).

Bucket aggregation. When the FPGA has processed all points, i.e. the first pass in-order and the following delayed passes, it starts the bucket aggregation phase. Here the curve adder pipeline is reconfigured to compute full `Add` operations as described in Section 4.3. These are 2-cycle operations with latency about $1.5T$, as the `mul` is about $1/2$ the length of the whole pipeline.

We can compare the latency of the bucket aggregation against the bucket accumulation phase, e.g. in the case of interest $N = 2^{26}$, $c = 16$, $T = 100$. Overall aggregation takes 2^c `Add` times c windows, i.e. 2^{20} `Add`, so the latency is about $1.5T \times 2^{20}$ stages. The total number of `MixedAdd` Nc is 2^{30}, hence the bucket aggregation phase accounts for about

[9]This is popular in software implementations, e.g. `gnark-crypto` and `arkworks`.

15% of the computation time.

We chose this approach to limit the complexity on the FPGA, sacrificing a bit on latency. In [Xav22], the authors show how to compute bucket aggregation in parallel at the price of a handful of extra `Add`. We believe a better approach would be to interleave bucket aggregation within the next window bucket accumulation, but we leave the implementation as future work. We also considered performing the bucket aggregation on the host, but this was ineffective on our platform.

Result aggregation. Finally, the result of each column is read by the host, and aggregated in the final result. As mentioned, this has a negligible cost. We preferred to leave this on the host side to minimize the complexity on the FPGA.

5 Evaluation

5.1 Methodology

We implemented the entire system in SystemVerilog RTL and evaluated it on AWS F1 with an `f1.2xlarge` instance. This cloud platform provides access to cards with AMD-Xilinx VU9P FPGAs. The card has four independent DDR channels, each with 16GB capacity and a theoretical throughput of 16GB/s. The interface exposed to the design is 512-bits wide, and is saturated at 250MHz. On this platform, we rely on the shell provided by Amazon for all communications with the host and DDR. We find the platform and shell easy to use, taking about 20% of the VU9P chip's resources.

We report four metrics:

- **Resource utilization:** we report LUTs, registers, BRAMs, URAMs (if any), and DSPs used in each configuration.
- **Power:** we rely on Vivado's power report to get an estimate of the power dissipation of the design. It should be noted that these are estimates by the tool, and actual energy consumption can vary at runtime due to input data and environmental properties, and would require live measurements.
- **Clock speed:** we report the clock speed achieved for each design.
- **Latency:** we report the time it takes to process the entire MSM network, including the software component running on the host. We include data transfer times between the FPGA and the host.

5.2 Experimental Results

Our design works at 250MHz. We stress that the numbers reported in this section strictly depend on the clock speed. For example, the length of the curve adder pipeline is 96 stages, but if we compile our design at 125MHz it lowers to 65 stages.

Table 2 reports resource utilization for our design. MSM refers to the full system, including the AWS shell; `MixedAdd` is the curve adder pipeline only, mainly composed of 7 multipliers; `mul` is one multiplier, specifically the one used in the pipeline reconfiguration to achieve the full `Add`. All 7 multipliers are similar, but minor variations can occur at compilation time. At 250MHz, AWS reports that the max FPGA consumption is 51W, average 14W. Vivado tools estimated 77.69W for the whole system, and 43.37W for our design only.

In Table 3 we report timings to compute a full MSM with UR scalars, for various size of N. For comparison, we run `gnark-crypto` on a 18-core Intel Core i9-7980XE CPU running at 4.8GHz. On this machine power consumption is around 530W. This is the

Table 2: Utilization report for MSM.

System	LUTs	Regs	BRAM	URAM	DSP	Pipeline stages
MSM	525,298	661,146	404	219	2,277	–
`MixedAdd`	310,717	337,944	–	–	2,268	96
`mul`	43,144	46,866	–	–	324	39

fastest MSM benchmark we were able to achieve on CPU, while also testing `arkworks` on the same and other machines.

For $N = 2^{26}$, the target of the ZPrize competition, we obtain over 4× performance improvement, consuming 1/10 of the power.

We also included `gnark-crypto` times on the same AWS F1 machine, which is 4-core Intel Xeon E5-2686 running at 2.3GHz. This comparison is meant to highlight the relatively poor performance of the AWS F1 CPU, that explain some of the design choices we had to make to achieve our results. At the end of the day, while our work is primarily on hardware, we need a software component to interact with the FPGA.

Table 3: Time (ms) to compute MSM on `BLS12-377` for various N.

	CycloneMSM	`gnark-crypto`	
	AWS F1 FPGA	AWS F1	Intel i9
N	250MHz	4× 2.3GHz	18× 4.8GHz
2^{22}	817.9	10,544	1,727
2^{23}	1,133	19,064	3,330
2^{24}	1,761	36,180	6,192
2^{25}	3,016	70,953	11,891
2^{26}	5,656	143,985	24,013

5.3 Batch Affine

For completeness, we also implemented CycloneMSM in software, showcasing the use of affine coordinates and batch inversion.

Our implementation is based on `gnark-crypto`. As we aim to contribute back to the project, we focused on minimizing changes and only speed up the core of the MSM computation. In CycloneMSM, buckets are represented in affine coordinates, accumulation is done in batches of size T using batch inversion, and aggregation is done using extended Jacobian coordinates –as in the original code– but with mixed addditions.

In Table 4 we report timings to compute a full MSM with UR scalars, for various size of N. We compare results on the 4-core AWS F1 at 2.3GHz (this time the FPGA is not used), the 18-core Intel i9 at 4.8GHz and a 8-core Macbook Pro with Apple M1 Pro at 3.2GHz.

On all platform we achieve +10% speed improvement for large N, and over 20%+ on Mac. We stress that for smaller N the impact decreases, but it's still 3-5% for $N = 2^{20}$. Experimentally, we found that $T = 100$ for $c = 20$, and $T = 40$ for $c = 16$ are good default values. We didn't consider smaller values of c, as the number of conflicts grows and this approach becomes ineffective.

In software like in hardware, we implemented the delayed scheduler. Interestingly, we couldn't build an efficient greedy scheduler, so we leave it as a challenge for the future. The main reason is the excess of memory writes: the greedy scheduler needs to somehow keep track of when a bucket was last used, for example by updating a hashmap of size

Table 4: Time (ms) to compute MSM in software on `BLS12-377` for various N.

	M1 Pro			**AWS F1**			**Intel i9**		
N	`gnark`	Cyclone	Gain	`gnark`	Cyclone	Gain	`gnark`	Cyclone	Gain
2^{20}	1,459	1,362	7%	2,656	2,571	3%	481	467	3%
2^{21}	2,870	2,629	8%	5,226	5,054	3%	946	891	6%
2^{22}	6,214	4,790	23%	10,544	8,928	15%	1,727	1,667	3%
2^{23}	11,340	8,793	22%	19,064	16,481	14%	3,330	2,830	15%
2^{24}	21,623	16,711	23%	36,180	31,547	13%	6,192	5,334	14%
2^{25}	41,982	33,016	21%	70,953	63,040	11%	11,891	10,239	14%
2^{26}	85,647	65,900	23%	143,985	124,231	14%	24,013	20,322	15%

2^{c-1} (number of buckets). Our implementation of the greedy scheduler was faster than the original `gnark-crypto` only on Mac and AWS F1, but not on Intel i9, therefore we abandoned it. In contrast, in the delayed scheduler we can detect collisions locally in each batch, we don't need a hashmap and we can signficantly reduce the number of memory writes, leading to the results in Table 4.

6 Conclusion and Future Work

We present CycloneMSM, an implementation of MSM on `BLS12-377` accelerated via FPGA. The implementation is *optimized* to compute a fixed-base MSM on `BLS12-377` for uniformly random (UR) scalars, with lowest latency. We achieve 5.66s for $N = 2^{26}$, sub-second for $N = 2^{22}$.

The novelty of our architecture are a *fully pipelined curve adder* that runs at 250MHz, and a *scheduler* to reorder operations, maximize the usage of the curve adder and boost perfomance. We analyze the real-world case of UR scalars and present efficient schedulers suitable for hardware acceleration, as well as software implementation using affine coordinates with batch inversion. For completeness, we show that the software implementation achieves +10-20% performance improvement over `gnark-crypto`.

In future work, we plan to extend our FPGA implementation to different fields and curve, and explore larger values of N. Some of the open challenges we're left with are: 1) compute a $N = 2^{23}$ MSM in sub-second, maybe even $N = 2^{24}$ (though we don't think it's achievable at 250MHz); 2) compute very large MSM, $N = 2^{36}$ and beyond; 3) implement a more efficient scheduler, along the line of the greedy scheduler.

References

[AEHG22] Diego F Aranha, Youssef El Housni, and Aurore Guillevic. A survey of elliptic curves for proof systems. Cryptology ePrint Archive, Paper 2022/586, 2022. https://ia.cr/2022/586.

[BCG+18] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Paper 2018/962, 2018. https://ia.cr/2018/962.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrú. Gemini: Elastic snarks for diverse environments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 427–457. Springer, 2022.

[BDLO12] Daniel J Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In *International Conference on Cryptology in India*, pages 454–473. Springer, 2012.

[BL07] Daniel J Bernstein and Tanja Lange. Explicit-formulas database, 2007. https://www.hyperelliptic.org/EFD.

[DDQ07] Guerric Meurice De Dormale and Jean-Jacques Quisquater. High-speed hardware implementations of elliptic curve cryptography: A survey. *Journal of systems architecture*, 53(2-3):72–84, 2007.

[DKS09] Christophe Doche, David R Kohel, and Francesco Sica. Double-base number system for multi-scalar multiplications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 502–517. Springer, 2009.

[Gut20] Gus Gutoski. Multi-scalar multiplication: state of the art & new ideas, 2020. Presented at zkStudyClub: https://youtu.be/Bl5mQA7UL2I.

[HWCD08] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 326–343. Springer, 2008.

[LTD08] Duo Liu, Zhiyong Tan, and Yiqi Dai. New elliptic curve multi-scalar multiplication algorithm for a pair of integers to resist spa. In *International Conference on Information Security and Cryptology*, pages 253–264. Springer, 2008.

[MSZ21] Simon Masson, Antonio Sanso, and Zhenfei Zhang. Bandersnatch: a fast elliptic curve built over the BLS12-381 scalar field. Cryptology ePrint Archive, Paper 2021/1152, 2021. https://ia.cr/2021/1152.

[OS02] Katsuyuki Okeya and Kouichi Sakurai. Fast multi-scalar multiplication methods on elliptic curves with precomputation strategy using montgomery trick. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 564–578. Springer, 2002.

[OS03] Katsuyuki Okeya and Kouichi Sakurai. Use of montgomery trick in precomputation of multi-scalar multiplication in elliptic curve cryptosystems. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(1):98–112, 2003.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 258–263. IEEE Computer Society, 1976.

[SIM12] Vorapong Suppakitpaisarn, Hiroshi Imai, and Edahiro Masato. Fastest multi-scalar multiplication based on optimal double-base chains. In *World Congress on Internet Security (WorldCIS-2012)*, pages 93–98. IEEE, 2012.

[SS14] Nagaraja Shylashree and Venugopalachar Sridhar. Hardware realization of fast multi-scalar elliptic curve point multiplication by reducing the hamming weights over gf (p). *International Journal of Computer Network and Information Security*, 6(10):57–63, 2014.

[SZZG21] Da-Zhi Sun, Ji-Dong Zhong, Hong-De Zhang, and Xiang-Yu Guo. On multi-scalar multiplication algorithms for register-constrained environments. *Electronics*, 10(5):605, 2021.

[Xav22] Charles F. Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. Cryptology ePrint Archive, Paper 2022/999, 2022. https://ia.cr/2022/999.

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428. IEEE, 2021.