FPGA Acceleration of Multi-Scalar Multiplication

ZPrize 2022

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

1 Executive Summary

We present 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 uniform 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 software implementation using affine coordinates with batch inversion.

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

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 PCI 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 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 78W (design only: 43W). For comparison, a purely software implementation in `arkworks` takes about 71.8s on a multi-core Intel Xeon Platinum 8380 CPU at 2.30GHz.

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.

Our work is motivated by participation in the ZPrize[1], a competition among industry partners and academic institutions to accelerate primitives related to zero-knowledge

[1] https://zprize.io

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 their experimental results do not show performance that scales linearly.

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

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 architecture 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 $N = 2^{36}$ took about a week to compute. 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 applications that requires proof of very large circuits (e.g. zkVMs).

2 Background

2.1 Elliptic Curves, Twisted Edwards

Let $\mathbb{F}_q$ be the Galois field of order q. 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 $\mathbb{G} \leq E(\mathbb{F}_q)$ 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[2]:

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

Every Twisted Edwards curve is birationally equivalent to an elliptic curve in Weierstrass form. The converse only holds for some curves. Different coordinate systems have advantages for particular applications. For this implementation, we aimed to minimize costly divisions while reducing the number of multiplications for each curve operation.

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:

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

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

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.

Let's call *bucket* $\mathcal{B}_k$, $k \in [0..2^c)$, 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 $\mathcal{B}_k$:

$$S_k = \sum_{P \in \mathcal{B}_k} P \tag{1}$$

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, that leads to less memory and most importantly half time for bucket aggregation.
- Parallelize bucket aggregation, e.g. by splitting computation in even and odd and re-assemble.

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: N `MixedAdd` (could exclude points with reduced scalar $= 0$)
- 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`.

Algorithm 1 Bucket Method for Multi-Scalar Multiplication

function MSM_INIT(points)
  convert points, *e.g. to Edwards*
  initialize context ctx
  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)
    $R \leftarrow 2^c R + R^{(j)}$
  return R

function ACCUMULATE(j, ctx, scalars)
  for $i = 0$ to $N - 1$ **do**
    (k, sgn) $\leftarrow$ REDUCE(scalars[i], j)
    $P \leftarrow$ sgn $\cdot$ ctx.points[i]
    $S_k \leftarrow S_k + P$

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

3 Multi-Scalar Multiplication

Equation (1), 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[3], gnark-crypto[4]) 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.

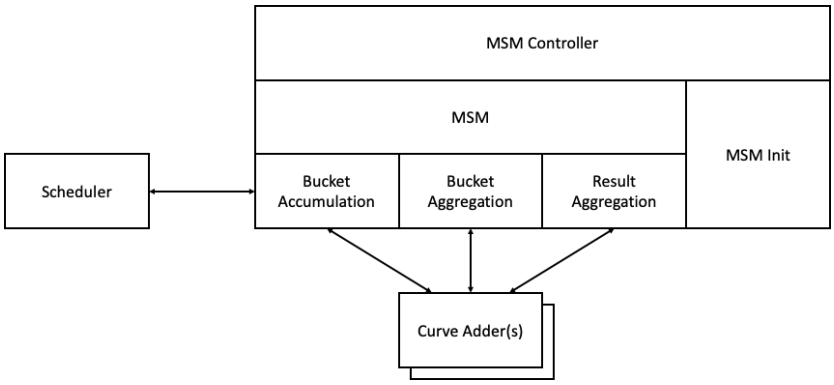

Figure 1: System architecture comprised of Curve Adder, MSM Controller and Scheduler.

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.

[3] https://github.com/arkworks-rs
[4] https://github.com/ConsenSys/gnark-crypto

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 in 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 an Curve Adder for actual computation. Bucket accumulation is driven by a Scheduler to maximize performance.

Scheduler. Responsible to optimize the order of operations during bucket aggregation, to maximize performance. We'll 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're 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:

$$k^{(t_1)} \neq k^{(t_2)} \qquad \text{if } |t_1 - t_2| \leq T \ . \tag{2}$$

The idea is that, when buckets are all different, the T additions can be performed more efficiently. Or vice-versa, if the buckets are all 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, it satisfies the condition (2); and 2) for efficiency, it only reorders a small percentage points and scalars.

If scalars are UR, so are the reduced scalars for every window. WLOG, let $\{\bar{n}_i\}_{i=0}^{N-1}$, $\bar{n}_i \leq 2^{c-1}$, a sequence of UR reduced scalars. Since N is large, we can approximate the sequence with a Poisson with $\lambda = N/2^{c-1}$. The probability to find a collision, i.e. $\bar{n}_j = \bar{n}_i$, with $i < j \leq i + T$ can be approximated as $NT/2^{c-1}$. For example, for $N = 2^{26}$, $c = 16$ and $T = 100$, we expect to find about $204,800$ collisions, which is about 3% of N.

Back to our original problem, so far we've seen that if we reschedule about 3% of the N points and reduced scalars (for each window), then condition (2) is satisfied. 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 $N = 2^{26}$ it's about 10, so the greedy scheduler requires very little storage compared to the delayed scheduler.

Proposed scheduler. The greedy scheduler enforces reprocessing a point as soon as possible. In our experience, simply relaxing the condition to "as soon as it's convenient" improves efficiency without any significant drawbacks. To illustrate the difference with respect to the greedy scheduler, consider the reduced scalars: $k_0 = 1,\ k_1 = 1,\ k_2 = 1,\ k_3 = 2,\ k_4 = 2,\ \ldots$ Then, $k^{(0)} = k_0,\ k^{(1)} = k_3$ (as 1 are conflicting), $k^{(T+1)} = k_1,\ k^{(T+2)} = k_4,\ k^{(2T+2)} = k_2$. So, k_4 needs to be re-ordered before k_3, i.e. somehow the queue of delayed points needs to maintain ordering. However, scheduling $k^{(T+1)} = k_1,\ k^{(2T+2)} = k_2,\ k^{(2T+3)} = k_4$ slightly reduces the complexity of managing the queue, and makes no appreciable difference in practice.

3.3 Applications: FPGA, Batch Affine

We consider two concrete applications that benefit from scheduling points in a different order: 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, in 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 doesn't conflict with buckets already inside the pipeline. This is equivalent to condition (2).

In software, the use of affine coordinates for bucket accumulation can provide optimal performance. Addition formulas in affine coordinates require 2 or 3 `mul` plus one field inversion. By computing additions in batches of T points and by using batch inversion, for large enough T the cost is reduced to 6 `mul` (worst case), which is better than any other coordinate system. In order to process additions in batch, the T input buckets need to be different, as expressed by condition (2).

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

4 FPGA Design

4.1 Field Arithmetic

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

Because of our choice of coordinates, we only need additions, subtractions and multiplications (`mul`) in the field (no costly divisions). 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 our explanation to our final implementation.

Our final field `mul` is built with three 384-bit integer multiplications. The first and last are done with 3-layer of 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 $ak \bmod R$, with k 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 k: we aggregated

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 use this fact to implement the curve adder and save some modular reductions.

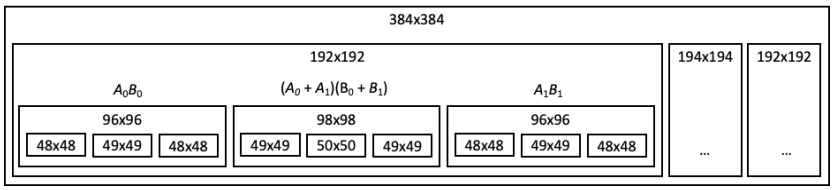

Figure 2: Integer multiplier, used to build field `mul`.

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 $a * b$, where a and b are represented as bits, accumulates the result of shifting a by i for ever 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 ones corresponding to the indices of b that are negative). The effect of this is that the depth of the adder trees is reduced in expectation from $log_2(pubcnt(b))$ to $log_3(pubcnt(b))$, where $pubcnt(b)$ is the hamming weight of b. The adder tree for a constant multiplier with NAF looks is shown in 3

Algorithm 2 NAF Multiplier

function MULTIPLYNAF(a, b)
  // b is constant
  $(b_0, \ldots, b_i \ldots, b_n) \leftarrow \text{bits}(b)$
  $\hat{R} \leftarrow NAF(R)$
  // 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 b is constant
    if b_i is 1 **then**
      $acc_{pos} \leftarrow (acc_{pos} + a \ll i)$
    if b_i is -1 **then**
      $acc_{neg} \leftarrow acc_{neg} + (a \ll i)$
  $acc \leftarrow acc_{pos} - acc_{neg}$
  return acc

We recursively apply Karatsuba's method to reduce the total number of multiplications by removing the partial results. Karatsuba's method is defined as follows: Given a radix R, and two integers a, b that can be decomposed into left and right components: $a = (a_0, a_1)$ $b = (b_0, b_1)$. For a decomposition of the form $(a_i, a_{i+1}, \ldots, a_n)$, let the

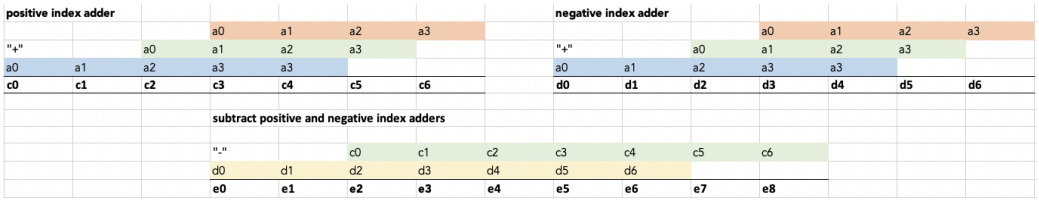

Figure 3: Adder Tree with NAF

right-most component a_n be the MSB.

$$\begin{aligned} a * b &= (a_1 R + a_0) * (b_1 R * b_0) = (a_1 b_1 R^2 + a_0 b_1 R + b_0 a_1 R + a_0 b_0) \\ &= a_1 b_1 R^2 + (a_0 b_1 + b_0 a_1) R + a_0 b_0 = x_2 R^2 + x_1 R + x_0 \\ &= (a_1 + a_0) * (b_1 + b_0) - x_2 - x_0 \end{aligned}$$

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[5]. The pipeline is shown in Figure 6, runs at 250MHz and has length 96 stages[6], so it can process a new `MixedAdd` every 4ns, returning the result after $96 \times 4\text{ns} = 384\text{ns}$.

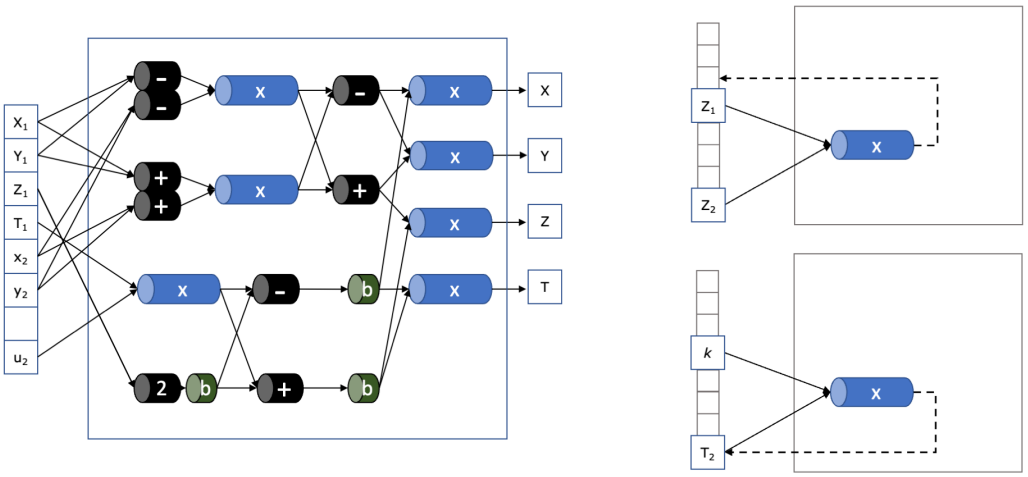

Figure 4: Fully pipelined `MixedAdd` in extended Twisted Edwards coordinates (left), where -, +, 2, $\times$ and `b` are resp. subtractions, additions, double, field multiplication and alignment buffers. The pipeline can be reconfigured to just run a field multiplication (right), leading to a multi-cycle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

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`. We used the 4-processor version of the formulas to reduce the pipeline length.

[5]We're using unified formulas, so computing `Dbl` or `Add` is the same.
[6]The length depends on the frequency, for example at 125MHz the length is 80 cycles.

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]

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 $a, b \in [0..q)$, $a + b$ 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`. To maintain low resource utilization we implemented it as a 2-cycle pipeline. Denoting $P_2 = (X_2 : Y_2 : Z_2 : T_2)$, at stage 0 we feed Z_1, Z_2 into the multiplier, at stage 1 we feed T_2, k. After 39 stages (the length of the multiplier) we retrieve the results $r_0 = Z_1 Z_2$ and $r_1 = kT_2$, and we 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.

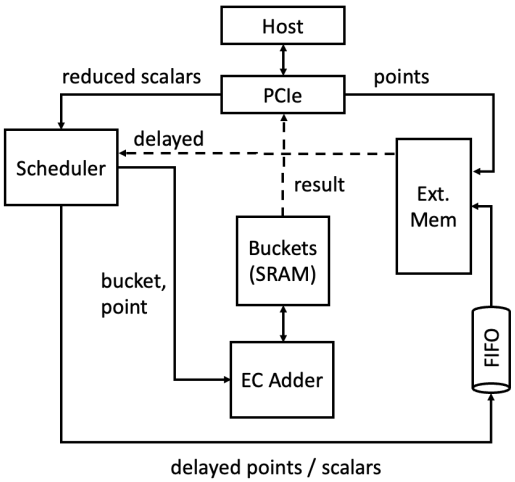

Figure 6: Architecture of FPGA implementation

MSM init. We assumed an initialization phase where inputs are N points in Weierstrass affine coordinates. Points are converted in affine Twisted Edwards $(x, y, u = kxy)$, sent to the FPGA and stored in DDR. Conversion from Weierstrass to Twisted Edwards costs approximately like a curve operation, and can be speed 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 cycle, i.e. at 250MHz we need to feed one reduced scalar every 4ns. Because of the PCI 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're sending one reduced scalar (16-bit) in every command. Commands are 64-bit long and we send them in batches of 8. We experimented with various protocols between host and FPGA, but in the final iteration commands are fundamentally only containing the reduced scalars. Scalars are sent in order, and the

Algorithm 3 Bucket Method for MSM 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)
    $R \leftarrow 2^c R + R^{(j)}$
  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 $\bar{R}$

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 didn't work in our experiments as we couldn't synthetize a pipeline at 250MHz.

Choice of window size c. The choice of $c = 16$ is primarily a tradeoff due to the amount of buckets and the speed of computing the reduced scalars. With $c = 17$ we estimated the whole algorithm would be only 7% faster (assuming a "perfect" implementation), but it'd require double the number of buckets and more computation on the host side.

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. Computing the reduced scalars on the fly gets slower for higher columns, again making it ineffective. Our latest implementation precomputes signed windows of 64 bits (i.e. it propagates carry at 64-bit level), and computes reduced scalars on the fly within these windows. This tradeoff maximizes the speed at which host can stream reduced scalars to the FPGA.

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 $S_k \leftarrow S_k + P_i$. 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. For $N = 2^{26}$ and UR scalars, for each column an average of about 235,000 points have conflicts. 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 processes at a 3rd pass. Practically speaking, given the hypothesis of UR scalars, it's very rare to ever get to a 4th pass, so this scheduler is pretty effective even if naive. 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. In this phase the curve adder pipeline is reconfigured to compute full `Add` operations (described in the previous section). Overall this take 2^c `Add`, i.e. about one million total `Add`. While these are 2-cycle operations, the total amount is low compared to the 30 million total `MixedAdd`. We have ideas on how to optimize this phase, but we leave them as future work. Finally, the result of each column is read by the host, and aggregated in the final result. As mentioned, this has a negligible cost.

5 Results

5.1 Methodology

We implemented the entire system in SystemVerilog RTL and have made the code publicly available on GitHub[7]. We value the reproducibility of all the results presented here, and chose platforms that are accessible to most designers.

We evaluate our implementation on AWS F1. 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 Amazon provided shell 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 1 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

[7] https://github.com/jumpcrypto

to achieve the full `Add`. All 7 multipliers are similar, but minor variations can occur at compilation time. At 250MHz, the whole system consumes 77.69W, and our design 43.37W.

Table 1: 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

In Table 2 we report timings to compute a full MSM with N UR scalars, for various size of N.

Table 2: Time to compute MSM, accelerated via FPGA

N	MSM (ms)
2^{22}	817.9
2^{23}	1,133
2^{24}	1,761
2^{25}	3,016
2^{26}	5,656

6 Conclusion and Future Work

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

The novelty of our architecture is a fully pipeline curve adder that runs at 250MHz, and scheduler to reorder operations and maximize performance, maximizing the usage of the curve adder. 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.

In future work, we plan to extend our FPGA implementation to different fields and curve, and explore larger values of N. For example, we plan to evaluate schedulers and provide an efficient implementation in affine coordinates, which are currently overlooked in most common software implementations.

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.

[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.