Hardcaml MSM: A High-Performance Split CPU-FPGA Multi-Scalar Multiplication Engine

Andy Ray
aray@janestreet.com
Jane Street
London, UK

Benjamin Devlin
bdevlin@janestreet.com
Jane Street
New York City, USA

Fu Yong Quah
fquah@janestreet.com
Jane Street
London, UK

Rahul Yesantharao
rayesantharao@janestreet.com
Jane Street
New York City, USA

ABSTRACT

This paper presents a split CPU-FPGA Multi-Scalar Multiplication (MSM) engine written in Hardcaml. Hardcaml MSM was submitted to the 2022 ZPrize cryptography competition and won 1st place in the FPGA track. Hardcaml MSM targets the BLS12-377 elliptic curve and is currently the lowest-latency implementation utilizing FPGAs published. For a MSM of order 2^{26} we achieve a single-round MSM latency of 5.518s and average power of 52W, with our design running at 278MHz. When performing multiple rounds of MSM with the same base points but random scalars, we are able to further mask host I/O and memory latency and reduce latency to 5.083s. This is a latency improvement of 13% over the previously fastest reported FPGA solution, and an improvement of 472% when compared to state of the art open-source CPU library gnark-crypto.

KEYWORDS

Zero-Knowledge Proofs, Multi-Scalar Multiplication, Eliptic Curve Cryptography, FPGA, RTL Domain-Specific Language

ACM Reference Format:
Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. 2024. Hardcaml MSM: A High-Performance Split CPU-FPGA Multi-Scalar Multiplication Engine. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626202.3637577

1 INTRODUCTION

Zero-knowledge proofs [7] (ZKPs) are powerful cryptography tools that allow a prover to prove that a certain statement is true without revealing any other information to the verifier. ZKPs are very attractive for applications where online privacy is paramount, for example digital signatures [10], online voting [19], blockchain [2], and distributed systems [23].

One class of ZKP getting a lot of attention recently is the Zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARK) [6]. This type of ZKP requires no interaction between the prover and verifier, and is compact and quick to verify.

One of the most popular zk-SNARK implementations, Groth16 [12], requires a huge number of elliptic curve (EC) operations known as Multi-Scalar Multiplications (MSM) and Number Theoretic Transforms (NTT). Current systems that use zk-SNARKs tend to require MSMs with millions of inputs. In this paper we focus on accelerating the MSM problem on these large-scale MSMs.

A crucial element of our success was making use of Hardcaml [21]. Hardcaml is an OCaml library that can be used to design and test hardware. Hardcaml leverages both the strong type system of OCaml, along with a verbose built-in circuit linter, to increase hardware design productivity, reliability, and efficiency. A built-in cycle-accurate simulator allows for unit level tests alongside the Hardcaml source code, which can optionally print digital ASCII waveforms. These tests provide fast feedback on designs and help catch future bugs.

To compute large MSMs efficiently, we need to optimize a number of aspects of our system, including elliptic curve primitives, higher-level MSM algorithms, and embedded system architecture. While prior work has presented both MSM and ZKP engines implemented in isolation on CPUs [8, 16], ASICs [25], GPUs [14, 17], FPGAs [1, 11, 24], etc, Hardcaml's robust design and testing capabilities allowed us to integrate these optimizations into a single design. As a result, Hardcaml MSM won first place in the ZPrize FPGA track [9]. Hardcaml MSM implements the BLS12-377 elliptic curve, and is available on GitHub[22]. We have tested Hardcaml MSM on a VU9P FPGA running on a split-CPU architecture offloading some tasks to the host CPU.

In summary, our contributions in this paper include:

- A fully pipelined, strongly unified mixed point adder that also supports subtraction and only requires 7 multiplication and 6 addition operations.
- System architecture and techniques to mask PCIe latency to increase performance.
- A stall controller that impacts performance by only 0.543% with simple heuristics that only require 4-deep FIFOs.
- A split CPU-FPGA architecture that allows for a more streamlined FPGA implementation acheiving 278MHz.

This work is licensed under a Creative Commons Attribution International 4.0 License.

FPGA '24, March 3–5, 2024, Monterey, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0418-5/24/03
https://doi.org/10.1145/3626202.3637577

2 PRIOR WORK

Performance of published ZKP accelerators has been steadily improving. One of the first FPGA solutions available on Amazon's AWS FPGA cloud [11] is designed for the ZCash blockchain. This performs better than a CPU implementation, but implements point multiplication naively and doesn't support large scale MSMs.

PipeZK [25] presents an ASIC ZKP accelerator, which targets both the MSM and NTT problems, but only shows results for MSMs up to order 2^{20}. They also do not attempt to map their accelerator to widely available FPGAs, which we believe poses a different set of challenges.

PipeMSM [24] presents a FPGA-based MSM accelerator, but only shows results for MSMs up to order 2^{20}. In addition, the point adder is implemented using projective coordinates and requires a larger number of multipliers and adders.

CycloneMSM [1] presents a FPGA-based MSM accelerator for MSMs up to order 2^{26}. While this design includes several novel optimizations, including a multi-cycle pipelined point adder also using Twisted Edwards coordinates, we present precomputations and new architecture they do not take advantage of.

A next-gen GPU implementation [14] shows higher performance than Hardcaml MSM, but our architecture introduces several new optimizations that are novel and beneficial.

3 THE MSM PROBLEM

In general, the Multi-Scalar Multiplication (MSM) problem is to take a list of scalars and points and compute the sum of each of the points scaled by its corresponding scalar, modulo a large λ-bit prime also known as its security-value, as shown in (1). Here N is the scale of the MSM, s_i is a δ-bit scalar, and p_i is an λ-bit EC point.

$$MSM = \sum_{i=1}^{N} p_i s_i \tag{1}$$

Elliptic curve cryptography (ECC) allows for smaller keys than non-EC cryptography such as RSA, while providing the same level of security. For example, a small 228-bit ECC key requires as much time to crack as a much larger 2,380-bit RSA key.

Implementing the MSM requires two elliptic curve operations: point addition and point doubling. These operations can be optimized with efficient modulo reduction algorithms such as Barrett [3] or Montgomery, and better-than-$O(n^2)$ multiplication techniques such as the Karatsuba [13] algorithm.

A naive algorithm using repeated point addition of scalar-point sums can compute MSMs with a few thousand inputs. When the scale of a MSM increases into the range of more than a million points, other algorithms such as Pippengers [20] provide much better performance. Pippengers is discussed in more detail in the architecture section. Figure 1 shows how the MSM problem relies on optimized EC primitives, and feeds into the overall ZKP algorithm.

4 HARDCAML MSM

Hardcaml MSM is a split CPU-FPGA accelerator written in Hardcaml and integrated into an Amazon AWS FPGA cloud F1 instance using Vitis. Hardcaml MSM won first place in the recent ZPrize competition, and at this time is the fastest reported FPGA solution to the large-scale MSM problem in accelerating ZKPs.

We target the BLS12-377 curve, which is popular in many ZKP systems due to its high security level, and which can be transformed into other coordinate systems that allow for faster hardware implementations. In BLS12-377 λ is 377 bits and δ is 253 bits.

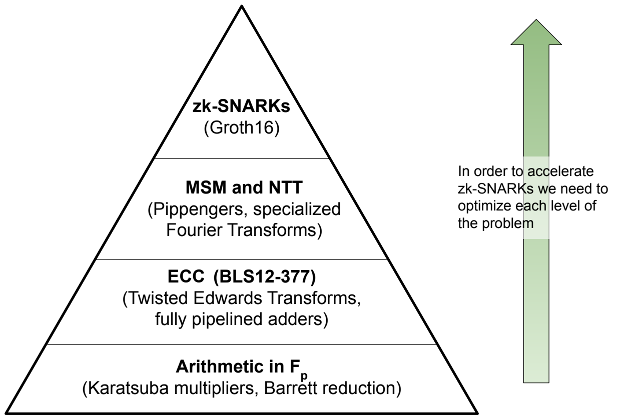
zk-SNARKs
(Groth16)
MSM and NTT
(Pippengers, specialized Fourier Transforms)
ECC (BLS12-377)
(Twisted Edwards Transforms, fully pipelined adders)
Arithmetic in F_p
(Karatsuba multipliers, Barrett reduction)
In order to accelerate zk-SNARKs we need to optimize each level of the problem

Figure 1: Breakdown of the ZKP algorithm into MSM, NTT, and EC primitives.

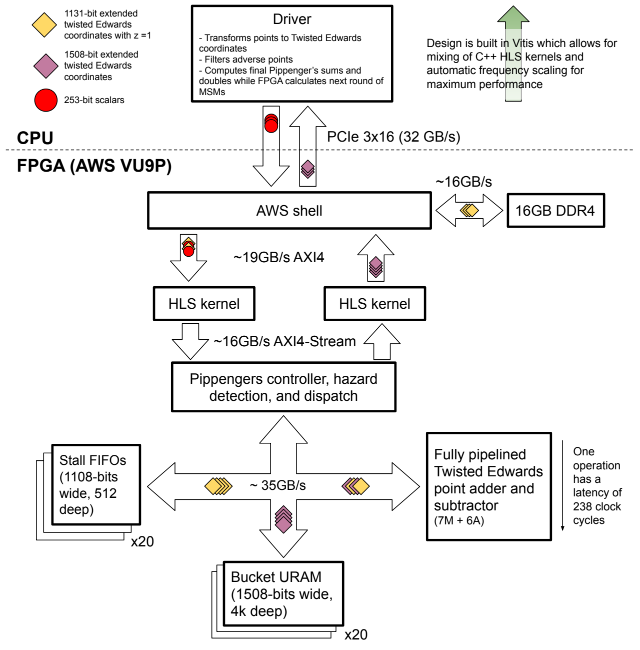
1131-bit extended twisted Edwards coordinates with z =1
1508-bit extended twisted Edwards coordinates
253-bit scalars
Driver
- Transforms points to Twisted Edwards coordinates
- Filters adverse points
- Computes final Pippenger's sums and doubles while FPGA calculates next round of MSMs
Design is built in Vitis which allows for mixing of C++ HLS kernels and automatic frequency scaling for maximum performance
CPU
PCIe 3x16 (32 GB/s)
FPGA (AWS VU9P)
~16GB/s
AWS shell
16GB DDR4
~19GB/s AXI4
HLS kernel
HLS kernel
~16GB/s AXI4-Stream
Pippengers controller, hazard detection, and dispatch
Stall FIFOs (1108-bits wide, 512 deep)
x20
~ 35GB/s
Fully pipelined Twisted Edwards point adder and subtractor (7M + 6A)
One operation has a latency of 238 clock cycles
Bucket URAM (1508-bits wide, 4k deep)
x20

Figure 2: Top-level architecture of Hardcaml MSM.

4.1 Top-Level Architecture

Figure 2 shows the top level architecture of Hardcaml MSM.

We implement a version of Pippenger's algorithm to solve the MSM problem. Pippenger's algorithm reformulates the dot products from (1) into smaller dot products over buckets, where each bucket represents a small contiguous slice of the scalar: $s_i[slice_w]$.

$$MSM = \sum_{w=0}^{W-1} 2^{wB} \left(\sum_{i=0}^{N-1} p_i s_i[slice_w] \right) \tag{2}$$

Here B is the bucket size in bits and W is the number of slices. p_i and s_i are elements of the prime and scalar fields respectively, and the product WB must be greater than or equal to δ, the number of bits in the scalar field.

The computational cost is broken down in (3) below, with the bucket sums, triangle sums, and final accumulation shown. Here **A** and **D** represent the cost of point additions and doubles respectively. To calculate the *bucket sums* $\sum_{i=0}^{N-1} p_i s_i[slice_w]$, for a given slice, we create 2^B buckets, one for each possible value of $s_i[slice_w]$. Then, in each $bucket_b$, we sum all the p_is such that $s_i[slice_w] = b$. One the host CPU, we calculate the *triangle sums* by multiplying each $bucket_b$ by b and adding them all together to get a sum for the slice. Finally, we combine the slice sums into the result of the MSM.

$$cost = (\underbrace{NA}_{\text{bucket sums}} + \underbrace{\frac{2^{B+1}A}{2}}_{\text{triangle sums}} + \underbrace{WD + A}_{\text{final accumulations}}) \frac{\delta}{W} \tag{3}$$

For large-scale MSMs ($N = 2^{20}$ and greater), the bulk of the computation lies in the bucket sums. For example, on the BLS12-377 curve with $N = 2^{26}$ and $W = 16$, the bucket sums require over 1 billion point additions, while the rest of the computation requires only about 8,000 operations. Because of this, we compute bucket sums on the FPGA, while in parallel performing triangle sums and final accumulations on the CPU.

Because the base points p do not change over multiple rounds of a a given ZKP, we can precompute curve and coordinate transformations on p and store them in DDR memory on the FPGA board. We then need to send only the scalars s to the FPGA to compute the MSM.

In order to pick values of bucket size B and number of slices W, we take into account the amount of memory used on the FPGA and the amount of time required to compute the triangle sums and final accumulations on the CPU. Previous work [1] reduced the number of bucket sums required by using a small number of large slices ($B = 16$). However, all the buckets for a slice of this size cannot fit in the FPGA's URAM. Values of $p_i s_i[slice_w]$ are not pre-sorted, as the scalars contain random values picked at runtime.

Instead, Hardcaml MSM uses a larger number of small slices ($B = 13$). This maps very well to Xilinx FPGA URAM memory primitives, such that each bucket is only 2 URAMs deep, can be floorplanned in such a way to allow for a high Fmax, and all bucket URAMs can fit onto the FPGA to be operated in parallel. On a VU9P this utilizes roughly 60% of the URAM available. Note that using $B = 16$ would mean that each bucket needs to be 16 URAMs deep, making it impossible to fit all bucket URAM on the FPGA, and limiting Fmax. Having all bucket URAM fit onto the FPGA allows us to completely parallelize the inner point additions, such that every $p_i s_i$ we stream into the FPGA can be processed immediately, without requiring sorting.

Depending on the depth of our mixed-point adder, we may submit an operation to a bucket that already has a pending operation in the adder pipeline. To solve this problem we implement a stall controller with FIFOs that can hold per-bucket $p_i s_i[slice_w]$ until that bucket is ready. We have provided modeling tools and found the throughput impact of this is only on average 0.543%, leading to very shallow FIFOs 4-deep that can be implemented logically in BRAM memory primitives.

The board we targeted has 64GiB of DRAM, split over 4 x DDR4 interfaces. Our point transformations only require 8.8GiB which allow us to localize access to a single interface. We implement FPGA-CPU communication using open-CL libraries in Vivado Vitis. For the interface between our Hardcaml MSM core and AWS shell on the FPGA, we make use of HLS kernels that allow frequency scaling and merging of data streams as they arrives from DRAM and PCIe. Rather than using a fixed clock, Vitis will attempt to find the maximum frequency possible, and then scale the provided clock post-implementation to allow for the highest Fmax.

4.2 Twisted Edwards Coordinates and Precomputation

The BLS12-377 curve has fixed parameters a and b. Affine points (p_x, p_y) in Weierstrass form on the curve follow the form

$$p_y^2 = p_x^3 + ap_x + b$$

Point addition in this form requires expensive inversions that we want to avoid. This section shows transformations onto a Scaled Twisted Edwards curve that will lead to very efficient hardware point addition operations.

First, we convert points in Weierstrass form to points on a Montgomery [18] curve, with the following formula (A and B are curve parameters):

$$By^2 = x^3 + Ax^2 + x \tag{4}$$

An elliptic curve in Weierstrass form is equivalent to a Montgomery curve with $A = 3\alpha s$ and $B = s$, where $s = \sqrt{3a^2 + a}^{-1}$ and α is one of the roots of $x^3 + ax + b = 0$.

A Twisted Edwards curve has the following formula, where a and d are parameters of the curve.

$$\begin{gathered} ax^2 + y^2 = 1 + dx^2y^2 \\ a = \frac{A+2}{B} \quad , \quad d = \frac{A-2}{B} \end{gathered} \tag{5}$$

Once we have this curve transformation we can convert points onto the Twisted Edwards curve [4] by

$$\begin{gathered} x_{mont} = s(x_{weierstrass} - a) \quad , \quad y_{mont} = sy_{weierstrass} \\ x_{twist} = \frac{x_{mont}}{y_{mont}} \quad , \quad y_{twist} = \frac{x_{mont} - 1}{x_{mont} + 1} \end{gathered} \tag{6}$$

The catch here is that not all points can be mapped–points with $x_{mont} = -1$ or $y_{mont} = 0$ are not valid on a Twisted Edwards curve, and there are 5 such adversarial points on the BLS12-377 curve. Because of our split CPU-FPGA architecture, we are able to filter these out on the host and if we encounter them, the portion of the MSM result they contribute to is calculated by a CPU side-band process without transformation.

While prior art [1] takes advantage of this transform, we take it further by transforming the curve into a Scaled Twisted Edwards Curve [5], given by

$$x_{scaled} = \left(\sqrt{\frac{-3as-2}{s}}\right)x_{twist} \quad , \quad y_{scaled} = y_{twist} \tag{7}$$

Another transformation unique to Hardcaml MSM is as follows. We take our points on the Scaled Twisted Edwards Curve in affine coordinates and transform them into Extended coordinates (8) which contain redundant values z and t but allow for a shorter point addition formula.

$$\begin{aligned} x_{ext} &= \frac{y_{aff} - x_{aff}}{2} \quad , \quad y_{ext} = \frac{y_{aff} + x_{aff}}{2} \\ t_{ext} &= 4dx_{aff}y_{aff} \quad , \quad z_{ext} = 0 \end{aligned} \tag{8}$$

Table 1 compares results of the number of field operations required after applying our transformations. Here **M**, **S**, **D**, and **A** represent the number of field multiplies, squares, multiplication by a constant, and additions or subtractions respectively.

When compared to prior work, our implementation achieves the smallest amount of field operations with only 7M and 6A. The previous best Scaled Twisted Edwards implementation, from the Explicit-Formulas Database [5] (EFD), achieves 7M + 8A + 2D, but it is not strongly unified, meaning doubling is not supported and special care would need to be taken to avoid the case when two points are identical.

4.3 Triangle Sum Offloading

After our controller has detected that a given bucket has finished processing all $p_i s_i[slice_w]$, it starts streaming the bucket values back to the host CPU. As the host is receiving these, it starts both the triangle sums and final accumulations for each bucket, and starts sending new s to the FPGA for further MSM rounds. Transferring data while both the CPU and FPGA are doing meaningful work masks the PCIe I/O penalty. Because we use many small buckets rather than a few large buckets, we can simultaneously calculate a new MSM once one bucket has been sent to the host, before all buckets have finished. We also do not rely on a fast CPU, as the triangle sums are much easier to calculate than the bucket sums. After the triangle sums are finished, the host CPU reverses the Scaled Twisted Edward transforms and sums in any adversarial $p_i s_i$ that it has stored.

4.4 Streaming Scalar Transformation

Kernels written in C++ and transformed via Vivado HLS combine the ps in the FPGA's DDR memory and the ss streaming from the CPU into $p_i s_i$, allowing the FPGA to start work as soon as it receives the first input.

In order to reduce the number of buckets required we perform a transformation on s on the FPGA in realtime. Each $s_i[slice_w]$ represents an unsigned integer in the range $[0, 2^b - 1]$. We can transform $s_i[slice_w]$ into the signed range $[-2^{b-1}, 2^{b-1} - 1]$ by subtracting 2^b and propagating the carry for each $s_i[slice_w]$ as it streams into the FPGA. We can exploit this because our mixed-point adder implements subtraction cheaply and all $s_i[slice_w] < 0$ can be added into positive bucket URAM slots by instead subtracting its value, meaning a 2^{13} bucket can be implemented by a 2^{12}-deep URAM.

4.5 Fully Pipelined Unified Point Adder

Our 4-stage unified mixed-point adder takes advantage of the previous optimizations to allow for a high Fmax of 278MHz, shown in Figure 3. Our fully-pipelined adder can accept new input every clock cycle and has a latency of 238 clock cycles. Both input points A and B are in extended coordinates; point B inputs originate from DRAM, converted from affine coordinates, and have a fixed initial z_{ext} coordinate of zero. Pipe blocks pipeline data that is not being modified in that stage. We provide an input *subtract or add* that when set will perform a $P_A - P_B$ operation instead of addition, required due to our signed scalar transformation. When performing subtraction, the first-stage multiplication and second-stage subtractions have their operands switched. Instead of doing a full reduction to $[\lambda - 1, 0]$ for each multiplication, we selectively perform a cheaper coarse modular reduction using Barrett reduction after the first and third stages.

4.6 Stall Controller

Figure 4 shows our stall controller and hazard detection dataflow. The hazard we must avoid is while a pair of points for a target bucket are being added together, any other points for the same bucket would corrupt its data. We implemented a stall controller and scalar shift register for the purpose of tracking which buckets could be processed, along with stall FIFOs to temporarily hold points in the case a bucket was busy. Our experiments showed that a 4-stage pipeline maximized our Fmax without a significant worsening of latency or stall rate.

Our stall controller tries to keep the point adder as busy as possible with a few simple heuristics. We need to track if data is present in any of the 238 pipelines and check if a new coefficient would cause a hazard. Naively done, this would require 238 comparators in parallel and then a wide OR reduction for each pipeline in the adder that can potentially contain valid data, which would impact Fmax.

4.6.1 Scalar Tracking. Our stall controller processes multiple slices on successive clock cycles rather than all in parallel. This means that for $slice_w$, at clock cycle w, we only check hazards for its buckets. The locations in the point adder with data from the same-slice are deterministic. This architecture means we only need to compare and logically `OR` reduce at most 12 of the 253-bit scalars in the pipeline rather than all 238.

4.6.2 Stall Point FIFO. When we detect a hazard the p_i and W-bit scalar slice $s_i[slice_w]$ are placed in a stalled point FIFO and we insert a bubble into the pipeline for that cycle. There are separate FIFOs for each bucket being processed, and because we only need to process one stall FIFO per clock cycle, all stall FIFOs can actually be logically mapped to a single wide BRAM.

Table 1: Number of field operations for a point addition.

	Curve	Coordinate System	Adder Type	Number of Field Operations
Hardcaml MSM	Scaled Twisted Edwards	Extended	Strongly unified mixed	7M + 6A
Cyclone MSM [1]	Twisted Edwards	Extended	Strongly unified mixed Strongly unified	7M + 8A +1D 9M + 8A +1D
Pipe MSM [24]	Twisted Edwards	Projective	Strongly unified	12M + 17A + 3D
EFD [5]	Weierstrass Scaled Twisted Edwards	Projective	Add only Strongly unfied	7M + 4S + 9A + 4D 7M +8A + 2D

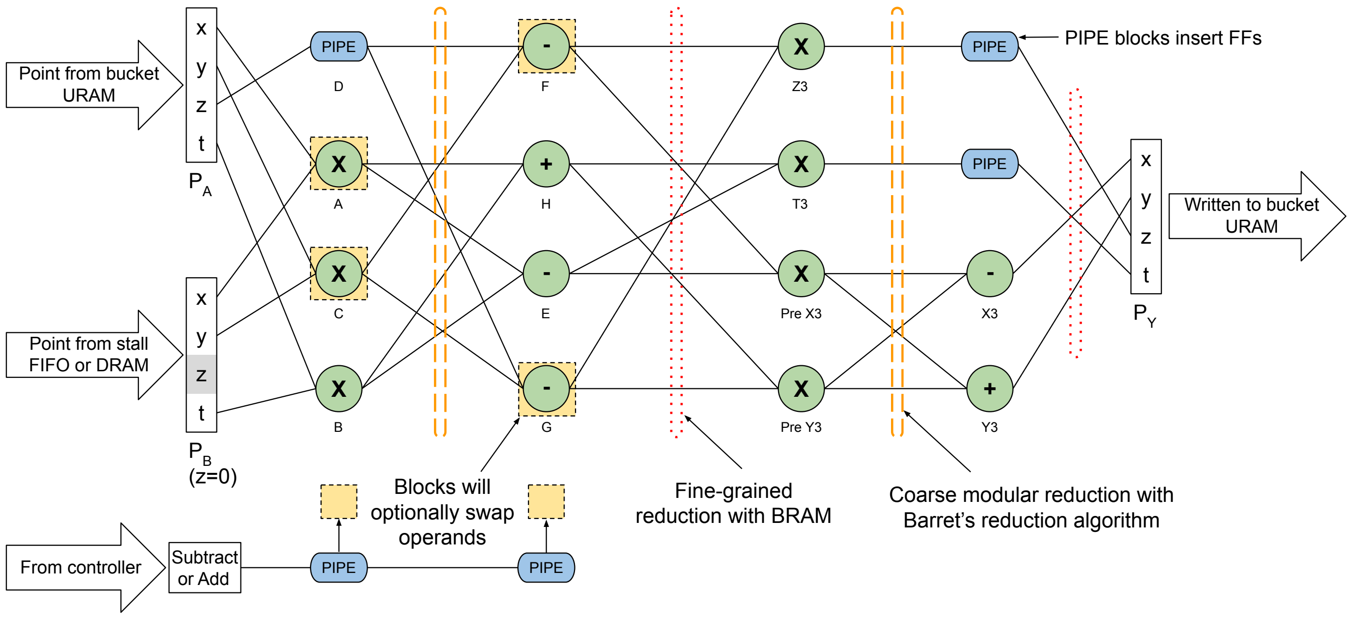

Figure 3: 4-Stage fully-pipelined unified mixed-point adder.

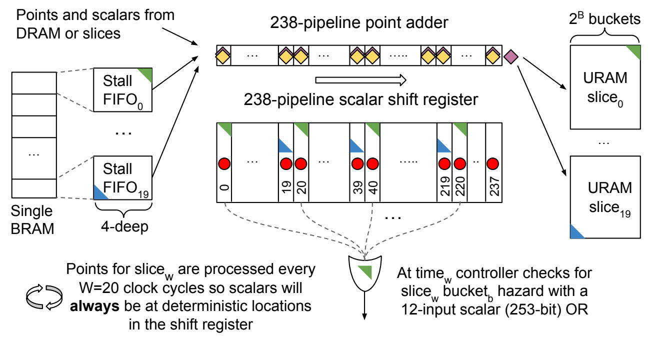
Points and scalars from DRAM or slices
238-pipeline point adder
2^B buckets
238-pipeline scalar shift register
Stall $FIFO_0$
Stall $FIFO_{19}$
Single BRAM
4-deep
URAM $slice_0$
URAM $slice_{19}$
0
19
20
39
40
219
220
237
Points for $slice_w$ are processed every W=20 clock cycles so scalars will **always** be at deterministic locations in the shift register
At $time_w$ controller checks for $slice_w$ $bucket_b$ hazard with a 12-input scalar (253-bit) OR

Figure 4: Dataflow of points and scalar tracking for hazard detection.

4.6.3 Heuristics. We modeled the stall controller and adder so that we could experiment with several algorithms to find the most efficient one. The algorithm we found that was the best was:

(1) If all the stall FIFOs have a least one $p_i s_i[slice_w]$, process them. Else,
(2) If any of the stalled point FIFOs are full, process them. Else,
(3) Process incoming $p_i s_i[slice_w]$

By processing full FIFOs first, we avoid overflow. When full, however, they must be flushed. We found that with only a 4-deep FIFO this was extremely rare. Modeling with this algorithm showed the performance drop due to stalls was only on average 0.543%.

4.7 Optimized Field Operations

The most costly operations inside our point adder are the field multiplications. In order to optimize these we use the Karatsuba algorithm, which requires $O(n^{log_2 3}) \approx O(n^{1.585})$ single-digit multiplications to multiply an n digit number, rather than the naive long-multiplication algorithm, which requires $O(n^2)$ single digit multiplications.

All multiplications are using modular arithmetic, so we chose Barret's reduction algorithm with slight modifications to improve the FPGA resource usage and allow a higher Fmax. We split Barrett reduction into the coarse reduction which is used after multiplication stages, and then fine-grained reduction after addition or subtraction stages. We are able to avoid costly multipliers in the fine-grained reduction step by storing reduction values in BRAMs [15].

We also replace any multiply-by-constant with one represented in non-adjacent form (NAF). If the constant has a Hamming weight in NAF larger than a certain threshold, we use DSP slices. Otherwise, we use long multiplication with LUTs. We also found that congestion was often the cause for lower Fmax, and by selecting Vivado place-and-route strategies that avoided congestion, and strategically applying location constraints to the point adder and bucket URAM, we increased post-route Fmax by nearly 10%.

Table 2: FPGA resource utilization on the AWS F1 VU9P.

	LUTs (k)	FFs (k)	BRAMs	URAMs	DSPs	DDR intf
Hardcaml MSM	388	731	419	463	2,999	1
Cyclone MSM [1]	525	661	404	219	2,277	3

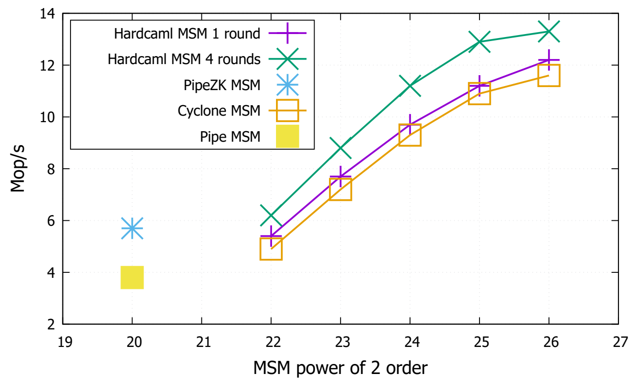

Figure 5: Performance to compute a MSM on BLS12-377 for different implementations.

5 MEASUREMENT RESULTS

We measured Hardcaml MSM on an AWS f1.2xlarge instance, targeting the BLS12-377 curve with the G1 subgroup-generator. This AWS instance contains a Intel Xeon E5-2686 v4 Processor (2.3 GHz (base) and 2.7 GHz (turbo)) and a UltraScale+ VU9P FPGA.

Table 2 shows resource usage compared to Cyclone MSM as it was the only other FPGA MSM accelerator implemented on AWS. Resource counts include the overhead from the AWS shell, roughly 20%. Our implementation favored using DSPs and FFs for point addition while Cyclone MSM uses more LUTs. The increased URAM is due to Hardcaml MSM mapping all buckets to the FPGA, while Cyclone MSM chose to have a larger B=16 but then only a single bucket mapped physically to the FPGA.

MSM performance was measured and shown in Figure 5 for both single-round and four-round power of 2 scale MSMs. We present Mop/s as a normalized performance number for comparison, although this does not take into account technology node. Here Cyclone MSM is implemented on the same VU9P 16nm FPGA, a 2^{26} MSM is reported as 11.7 Mop/s, compared to 12.1 Mop/s for Hardcaml MSM when computing a single round. PipeZK is an 28nm ASIC and acheives 5.7 Mop/s for a 2^{20} MSM. Pipe MSM is a U55C 16nm FPGA, and acheives 3.8 Mop/s for a 2^{20} MSM.

Table 3: Breakdown of time take in each step in Hardcaml MSM for $N = 2^{26}$.

Step	Time (ms.)
Memcpy 2^{26} scalars to special memory region	289
Transferring 2^{26} scalars to FPGA	198
Computing bucket point adds on FPGA	4968
Copying bucket values back from FPGA	1
Doing on-host bucket sums and post-processing	470

When we measure the time taken over multiple MSM rounds, we see the benefits of our split architecture, and memory and PCIe I/O optimizations. A four-round 2^{26} MSM takes 20.331s, which per-round gives a latency of 5.083s, and performance of 13.2Mop/s, which is a 13% improvement over previous state-of-the-art FPGA accelerators. As far as we can tell, prior work does not mask memory and PCIe I/O and does not gain any benefit when multiple rounds are performed.

It is not easy to do a apples to apples comparison to GPU implementations as they are implemented on different technology nodes, which was also highlighted by the ZPrize running separate FPGA and GPU tracks. Compared to the start-of-the-art GPU MSM accelerator by Matter Labs & Yrrid [14] running on an NVIDIA A40 we are 9x slower, although use 2.8x less power. For comparison the A40 was built on a process node 4 generations ahead of the FPGA we were given to implement against.

Although we implemented a split CPU-FPGA architecture, the bulk of the time is still consumed by the FPGA. Table 3 shows the breakdown of the time taken in each step. Note that the total time is not the sum of these as stages of each are happening in parallel.

6 CONCLUSIONS

This paper presents Hardcaml MSM, a split CPU-FPGA MSM engine written in Hardcaml, with the highest performance currently published for an FPGA implementation targeting the BLS12-377 curve. Hardcaml won first place in the FPGA track of the 2022 ZPrize cryptography competition. For a MSM of order 2^{26} we achieve a single-round MSM latency of 5.518s and average power of 52W, with our design running at 278MHz. When performing multiple rounds of MSM with the same base points but random scalars, we are able to further mask host I/O and memory latency and reduce latency to 5.083s. This is a latency improvement of 13% over the previously fastest reported FPGA solution [1], and an improvement of 472% when compared to state-of-the-art open-source CPU library gnark-crypto [8].

ACKNOWLEDGMENTS

The authors would like to thank Pranjal Vachaspati for constructive criticism and proofreading of the manuscript.

REFERENCES

[1] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. 2022. FPGA Acceleration of Multi-Scalar Multiplication: CycloneMSM. Cryptology ePrint Archive, Paper 2022/1396. https://eprint.iacr.org/2022/1396 https://eprint.iacr.org/2022/1396.

[2] Aritra Banerjee, Michael Clear, and Hitesh Tewari. 2020. Demystifying the Role of zk-SNARKs in Zcash. In *2020 IEEE Conference on Application, Information and Network Security (AINS)*. 12–19. https://doi.org/10.1109/AINS50155.2020.9315064

[3] Paul Barrett. 1987. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO' 86*, Andrew M. Odlyzko (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.

[4] Daniel J. Bernstein and Tanja Lange. 2017. Montgomery curves and the Montgomery ladder. Cryptology ePrint Archive, Paper 2017/293. https://eprint.iacr.org/2017/293 https://eprint.iacr.org/2017/293.

[5] Daniel J. Bernstein and Tanja Langea. 2007. Explicit-formulas database. https://hyperelliptic.org/EFD

[6] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again *(ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 326–349. https://doi.org/10.1145/2090236.2090263

[7] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-Interactive Zero-Knowledge and Its Applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing* (Chicago, Illinois, USA) *(STOC '88)*. Association for Computing Machinery, New York, NY, USA, 103–112. https://doi.org/10.1145/62212.62222

[8] Gautam Botrel, Thomas Piellard, Youssef El Housni, Arya Tabaie, Gus Gutoski, and Ivo Kubjas. 2023. ConsenSys/gnark-crypto: v0.9.0. https://doi.org/10.5281/zenodo.5815453

[9] ZPrize Commity. 2023. ZPrize. https://www.zprize.io/

[10] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. 2016. Cinderella: Turning Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation. In *2016 IEEE Symposium on Security and Privacy (SP)*. 235–254. https://doi.org/10.1109/SP.2016.22

[11] Benjamin Devlin. 2022. ZCash FPGA Accellerator. https://github.com/ZcashFoundation/zcash-fpga

[12] Jens Groth. 2016. On the Size of Pairing-Based Non-Interactive Arguments. In *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*. Springer-Verlag, Berlin, Heidelberg, 305–326.

[13] A Karatsuba and Yu. Ofman. 1962. Multiplication of many-digital numbers by automatic computers. In *Dokl. Akad. Nauk SSSR - Volume 145*. 293–3294.

[14] Matter Labs and Yrrid. 2023. ZPrize MSM on the GPU. https://github.com/matter-labs/z-prize-msm-gpu-combined

[15] Martin Langhammer and Bogdan Pasca. 2021. Efficient FPGA Modular Multiplication Implementation. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 217–223. https://doi.org/10.1145/3431920.3439306

[16] Guiwen Luo and Guang Gong. 2023. Fast Computation of Multi-Scalar Multiplication for Pairing-Based zkSNARK Applications. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–5. https://doi.org/10.1109/ICBC56567.2023.10174952

[17] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU Accelerated Zero-Knowledge Proof System. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/3575693.3575711

[18] Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48 (1987), 243–264.

[19] Malik Hamza Murtaza, Zahoor Ahmed Alizai, and Zubair Iqbal. 2019. Blockchain Based Anonymous Voting System Using zkSNARKs. In *2019 International Conference on Applied and Engineering Mathematics (ICAEM)*. 209–214. https://doi.org/10.1109/ICAEM.2019.8853836

[20] Nicholas Pippenger. 1980. On the Evaluation of Powers and Monomials. *SIAM J. Comput.* 9, 2 (1980), 230–250. https://doi.org/10.1137/0209022 arXiv:https://doi.org/10.1137/0209022

[21] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. 2023. Hardcaml: An OCaml Hardware Domain-Specific Language for Efficient and Robust Design. arXiv:2312.15035 [cs.PL]

[22] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. 2023. *Hardcaml MSM*. https://doi.org/10.5281/zenodo.10278449

[23] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca A. Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero Knowledge Proof System. In *IACR Cryptology ePrint Archive*.

[24] Charles. F. Xavier. 2022. PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication. Cryptology ePrint Archive, Paper 2022/999. https://eprint.iacr.org/2022/999 https://eprint.iacr.org/2022/999.

[25] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 416–428. https://doi.org/10.1109/ISCA52012.2021.00040