Efficient Parallelization of Large-Scale Modular Multiplication via Low-Latency LogJumps

Short Paper

Selim Kırbıyık, Maciej Czuprynko, Florian Krieger, Florian Hirner, Sujoy Sinha Roy
Institute of Information Security, Graz University of Technology, Austria
{*first.last*}*@tugraz.at*

***Abstract*—Elliptic-Curve Cryptography (ECC) found in Zero-Knowledge Proofs (ZKPs) protects assets worth more than a billion dollars on privacy-preserving blockchain networks. There, the transaction rate is mostly limited by the computational cost of Multi-Scalar Multiplications (MSMs). Thus, hardware acceleration of these operations, for instance, using FPGAs, is of interest. Current accelerators leverage the Pippenger algorithm to compute the MSMs. Due to data dependencies, the algorithm's performance is affected by the latency of the internal modular multipliers. Typically, these multiplications are realized using Montgomery multipliers. A promising variant of this algorithm is LogJumps which offers potential for parallelism and lower latency. This is achieved by reducing data dependencies within the Montgomery reduction. Yet, prior work has neither formalized nor explored a low-latency, hardware-friendly parallel realization of this method. We address this gap by formalizing parallelism in LogJumps and presenting the first practical, truly parallel LogJumps architecture for the BLS12-377 prime. Our modular multiplication design achieves up to 56% lower latency than the lowest-latency multiplier in the ZPRIZE 2022, while maintaining a 250 MHz frequency and throughput. Furthermore, we reduce the MSM latency by up to 1.85× using a full-point adder pipeline, while our logic consumption increases by only 1.32×.**

***Index Terms*—Reduction, ZKP, MSM, FPGA, Arithmetic.**

I. Introduction

Zero Knowledge Proofs (ZKPs) are powerful cryptographic primitives that can prove the knowledge of a secret to another party without disclosing the secret itself. This unique property of ZKPs attracts broad interest in academia and industry, with applications in verifiable cloud computing [1], online voting [2], and anonymous blockchain transactions [3]. For example, the Zcash cryptocurrency uses ZKPs in widespread production, wherein ZKPs protect assets worth billions of dollars [4]. Although practical deployments exist, ZKPs still suffer from high computational complexity. In particular, the Multi-Scalar Multiplication (MSM) involved in many state-of-the-art ZKP constructions [5]–[7] is a critical performance bottleneck and consumes more than 70% of the total proving runtime [8]. The large latency share of MSM is caused by its expensive Elliptic Curve (EC) operations over the BLS12-377 curve and its 377-bit-wide modular multiplications. These large operands complicate performant MSM computations.

This work has been supported in parts by the European Union's Horizon Europe research and innovation programme through the Marie Skłodowska-Curie grant under agreement 101226463, Austrian FWF grant PAT6402023, State Government of Styria, Austria, Department Zukunftsfonds Steiermark.

One efficient and widely used algorithm to compute MSM is Pippenger's method [9]. It splits MSM into three phases: bucket classification (BC), bucket aggregation (BA), and result aggregation (RA). In BC, the input elliptic-curve points are sorted into buckets according to the corresponding scalar window values. This phase contains many bucket-update operations, which can largely be processed *independently* in a pipelined point-adder architecture. Therefore, BC mainly benefits from high throughput. In contrast, BA and RA combine the bucket contents through *dependent* point additions and doublings. These phases are sequential and therefore benefit primarily from low-latency point-operation pipelines.

Building upon Pippenger's algorithm, existing hardware accelerators on FPGA [10], [11] and ASIC [8], [12] platforms mainly target the expensive BC step, which requires high point-addition throughput. Therefore, existing works instantiate several cascaded and highly pipelined 377-bit modular multipliers to form a fully pipelined point addition module.

However, the pipeline of these designs easily reaches 100 stages, which leads to two undesired effects. First, it increases the number of collisions in the BC step. These collisions occur if an input point must be added to a bucket currently processed in the deep point adder pipeline. In this case, the pipeline must be stalled until the target bucket is available. Second, the BA and RA cannot fill the pipeline due to the double and add approach. Again, the pipeline must be stalled since a new point operation can only be issued after the result of a previous operation is known. Both cases incur a performance penalty due to pipeline stalls, and the number of stalls is *proportional to the pipeline latency*. Hence, shorter pipelines with fewer stages are beneficial to accelerate Pippenger's algorithm. However, fewer pipeline stages and high clock frequency – essential for performant BC – are contradictory aspects in hardware design.

In this work, we address this problem by proposing a low-latency modular multiplier for FPGA-based MSM acceleration. The main contributions of this paper are as follows:

- We formalize the parallel LogJumps approach for Montgomery reduction and present its first practical hardware implementation as a parametric SystemVerilog RTL generator for FPGA platforms.[1]

[1] https://github.com/threonyl/fpl-artifact-modular-multiplier

- We optimize the proposed design for large-prime modular multiplication for BLS12-377/381. Our modular multiplier reduces pipeline latency by 1.29× and up to 2.29× compared to state-of-the-art FPGA designs.
- We integrate the proposed modular multiplier into a point-adder pipeline for MSM. This reduces bucket-aggregation latency by up to 1.85× and improves total MSM latency by up to 1.51×, while exposing a clear trade-off between latency and area on FPGA.

II. Background

This section introduces the mathematical background.

A. MSM and Pippenger Algorithm

MSM is a performance-critical computation in pairing-based ZKP systems and accounts for more than 70% of prover runtime [8], [13]. Given elliptic-curve points P_i and scalars s_i, it computes the weighted sum $Q = \sum_{i=0}^{k-1} s_i P_i$. Practical systems involve up to $k = 2^{28}$ points, making MSM a major target for hardware acceleration. A widely used approach is Pippenger's bucket method, summarized in Algorithm 1. Each scalar is decomposed into m windows of width ω bits, such that $s_i = \sum_{j=0}^{m-1} s_{i,j} 2^{j\omega}$ with $s_{i,j} \in [0, 2^\omega - 1]$. The computation is organized into bucket classification (BC), bucket aggregation (BA), and result aggregation (RA). BC assigns points to buckets according to the current window value, BA accumulates the buckets of one window into a partial sum S_j, and RA combines all window sums into the final MSM result Q.

Latency sensitivity of the Pippenger phases. The three phases of Algorithm 1 differ significantly in their hardware behavior. BC mainly performs updates of the form $B_{j,b} \leftarrow B_{j,b} + P_i$ and is therefore dominated by independent EC point additions (PADD). These updates are typically implemented using fully pipelined point-adder datapaths to sustain high throughput [13]–[15]. However, long pipeline latency increases the cost of bucket conflicts, since a later point mapped to the same bucket must wait until the earlier update has sufficiently progressed through the pipeline. In contrast, BA and RA are dominated by data dependencies: bucket aggregation forms each S_j through dependent additions, while result aggregation iterates over windows using $Q \leftarrow 2^\omega Q + S_j$. Thus, BA is dominated by sequential PADDs, whereas RA requires both repeated point doublings (PDBL) and point additions. Since these phases are inherently serial, point-operation latency directly determines their runtime.

Implication for modular arithmetic. At the arithmetic level, both PADD and PDBL are realized by finite-field operations over a large prime field. In projective-style coordinates, inversions are avoided, but each point operation still requires several modular additions, squarings, and, most importantly, large-prime modular multiplications. Consequently, the latency of the modular multiplier directly affects the latency of the PADD/PDBL pipeline and therefore the runtime of the dependency-sensitive aggregation phases (BA and RA). This makes low-latency modular multiplication a key design objective for efficient Pippenger-based MSM accelerators.

Algorithm 1 Pippenger MSM (bucket form) [9]

Input: scalars s_i, points P_i, window width ω
Split each s_i into windows $s_{i,j}$ for $j = 0, \ldots, m-1$
for $j = 0$ to $m-1$ **do**
  Initialize buckets $B_{j,b} \leftarrow \mathcal{O}$ for $b = 0, \ldots, 2^\omega - 1$
  for $i = 0$ to $k-1$ **do**
    $b \leftarrow s_{i,j}$
    if $b \neq 0$ **then** $B_{j,b} \leftarrow B_{j,b} + P_i$ ▷ *BC*
  end for
  $S_j \leftarrow \sum_{b=1}^{2^\omega - 1} b \cdot B_{j,b}$ ▷ *BA*
end for
$Q \leftarrow \mathcal{O}$
for $j = m-1$ downto 0 **do**
  $Q \leftarrow 2^\omega \cdot Q + S_j$ ▷ *RA*
end for
return Q

Algorithm 2 Multi-Word Montgomery Red. (SOS) [16]

Require: $A = (A_{2n-1}, \ldots, A_1, A_0)$ in radix $R = 2^w$, $A < N \cdot R^n$, modulus N (odd, n limbs), $R^n > N$, $\mu = -N^{-1} \bmod R$
Ensure: $Z = A \cdot R^{-n} \bmod N$
1: $Z \leftarrow A$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $q_i \leftarrow Z_i \cdot \mu \bmod R, \quad C \leftarrow 0$
4: **for** $j = 0$ **to** $n-1$ **do**
5: $(C, Z_{i+j}) \leftarrow Z_{i+j} + q_i \cdot N_j + C$
6: **end for**
7: **for** $j = i+n$ **to** $2n$ **do**
8: $(C, Z_j) \leftarrow Z_j + C$
9: **end for**
10: **end for**
11: $Z \leftarrow (Z_{2n}, Z_{2n-1}, \ldots, Z_{n+1}, Z_n)$
12: **if** $Z \geq N$ **then** $Z \leftarrow Z - N$
13: **return** Z

B. Montgomery Reduction

Montgomery reduction is an efficient algorithm to compute modular reduction modulo N on hardware. Montgomery reduction requires that $\gcd(N, R) = 1$. We use $R = 2^w, w \in \mathbb{Z}_{>0}$; for an operand of n limbs the Montgomery factor is then R^n with $R^n > N$. Thereby, Montgomery reduction avoids slow generic division by exploiting the cheap divisions by powers of two on hardware. For large integer modular multiplications on hardware, we need multiprecision Montgomery multiplications. The work in [16] analyzes, categorizes, and presents multi-precision variants of Montgomery reduction. The Separate Operand Scanning (SOS) variant, as seen in Algorithm 2, separates multiplication and reduction into distinct stages of the algorithm. Furthermore, Algorithm 2 computes the reduction sequentially (loop in line 2), by reducing one limb per loop iteration.

C. Related Work

ZPRIZE[2] is a competition that brings together industry and academia to improve the performance of ZKP primitives, such

[2]https://www.zprize.io/

as MSMs. CycloneMSM [14] achieved the second-best result in the 2022 MSM acceleration competition; they implemented their work in RTL. Their work also implements Montgomery modular multiplication as in Algorithm 2. The latest competition winners are Yrrid and Ingonyama [17], who use HLS to synthesize their work and also use a Montgomery modular multiplier. Both feature a fully pipelined modular multiplier and an unrolled point adder.

III. Parallel LogJumps Architecture

In this section, we present the formal definition of the parallel LogJumps algorithm [18] and our hardware architecture optimized to exploit parallelism within the reduction operation. The key idea is to decouple multiplication and reduction as in SOS. Then reduce $n-1$ limbs *in parallel* rather than sequentially. We first formally present the algorithm in Sec. III-A, then we describe the architecture in Sec. III-B. Finally, we explore three optimizations: limb decomposition strategy in Sec. III-C, latency reduction in the modular accumulation tree in Sec. III-D, and parameterized EDA tooling in Sec. III-E. Throughout this section, we use BLS12-377 with word size $w = 17$ and $n = \lceil 377/17 \rceil = 23$ limbs.

A. Parallel LogJumps Algorithm

Precomputing R^{-1}. The parallel LogJumps algorithm is based on the observation that multiplying $A = A_1 \cdot R + A_0 < NR$ by $\rho = R^{-1} \pmod N$ preserves the following congruence $A\rho = AR^{-1} = A_1 + A_0\rho \pmod N$. Hence, multiplying A by ρ is congruent to dividing the lower limb A_0 by R (mod) N and adding the result to the higher limb A_1.

Breaking the carry chain using ρ. In the multiprecision SOS Montgomery reduction, the carries of previous stages mutate the individual limb values. So, the result is dependent on the carry resolution of the lower limbs as shown in Eq. (1).

$$\begin{aligned} A &= A_{2n-1}R^{2n-1} + \cdots + A_1R + A_0 \\ AR^{-1} &= A_{2n-1}R^{2n-2} + \cdots + A_1 + A_0R^{-1} \pmod N \\ AR^{-n} &= A'_{2n-1}R^{n-1} + \cdots + A'_n \pmod N \end{aligned} \tag{1}$$

To break this carry-chain dependency, we can defer carry resolution until the end. Hence, we precompute $\rho_i = R^{-i}$ corresponding to the least significant first $n-1$ limbs. The parallelization arises from multiplying the $n-1$ limbs *independently* by ρ_i in *parallel* and resolving carries *once*. This deferred carry resolution is the difference between sequential multiprecision Montgomery variants such as SOS and the novel LogJumps. The least significant n-th limb will be reduced at the end, after we resolve carries and perform a final regular multiprecision Montgomery reduction. We resolve the carries after multiplying with ρ_i as $T = \lfloor A/R^{n-1} \rfloor + \sum_{i=0}^{n-2} A_i\rho_{n-1-i}$.

Accumulation and bounds. For the final Montgomery reduction to work, the input must lie in $[0, NR)$. There are $n-1$ many $A_i\rho_{n-1-i}$ products, where each is bounded by $(R-1)(N-1) < NR$. Similarly, the input A is also bounded by $A < NR^n$ by definition. So $\lfloor A/R^{n-1} \rfloor < NR$. Our variant of LogJumps removes the multiples of NR by conditionally subtracting NR, $n-1$ times.

Algorithm 3 Parallel LogJumps Reduction [18]

Require: $A = (A_{2n-1}, \ldots, A_1, A_0)$ in radix $R = 2^w$, $A < N \cdot R^n$, modulus N (odd, n limbs), $\rho_i = R^{-i} \bmod N$ for $1 \le i \le n-1$, $\mu = -N^{-1} \bmod R$
Ensure: $Z = A \cdot R^{-n} \bmod N$
1: $A_{\text{lo}} \leftarrow (A_0, A_1, \ldots, A_{n-2}), \quad A_{\text{hi}} \leftarrow \lfloor A/R^{n-1} \rfloor$
2: $T \leftarrow A_{\text{hi}} + \sum_{i=0}^{n-2} A_{\text{lo}_i} \cdot \rho_{n-1-i}$
3: **repeat** $n-1$ **times: if** $T \ge N \cdot R$ **then** $T \leftarrow T - N \cdot R$
4: $q \leftarrow (T \bmod R) \cdot \mu \bmod R$
5: $Z \leftarrow (T + q \cdot N)/R$
6: **if** $Z \ge N$ **then** $Z \leftarrow Z - N$
7: **return** Z

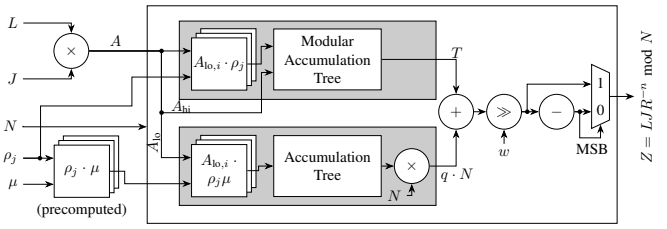

Fig. 1. Architecture overview of our LogJumps modular multiplier.

Final Montgomery reduction. Once the input is within $[0, NR)$ we apply a final regular multiprecision Montgomery reduction to bring the result into $[0, 2N)$. Then a single conditional subtraction at the end brings the result into the desired range of $[0, N)$. The full description of the algorithm is presented in Algorithm 3. Note that the operation in line 2 can be performed in parallel, which is not directly possible in the SOS variant.

Handling word sizes. We choose the smallest n such that $R^n > N$, and bound both inputs by $N-1$, yielding a conservative and correct bounds analysis.

B. Hardware Architecture

Fig. 1 shows the top-level dataflow of our LogJumps modular multiplier. The design consists of four major stages, each of which we now describe in detail. Pipeline registers are omitted in figures for clarity.

Stage 1: Limb routing and ρ-multiplication. The $2n$-limb input A is split into $n-1$ low limbs A_{lo} and the remaining $n+1$ high limbs A_{hi}. Each A_{lo} limb goes into a $w \times (\lceil \log_2(N+1) \rceil)$-bit multiplier that computes $A_{\text{lo}_i}\rho_{n-1-i}$. Hence, each product is a $(\lceil \log_2(N+1) \rceil + w)$-bit value. All $n-1$ multiplications execute simultaneously in a single pipeline stage. For BLS12-377, this amounts to 22 independent 17×377-bit parallel multiplications with 394-bit results.

Stage 2: Modular Accumulation Tree. The $n-1$ products together with A_{hi} are summed together using a Carry-Save-Adder (CSA) tree. We keep the intermediate results in carry-save form, avoiding full carry propagation until the very end. Only then, a single Carry-Propagate-Adder (CPA) converts the intermediate results back into binary form. The intermediate result is fed into the binary search conditional subtraction circuit, where the multiples of NR are subtracted, producing the final full-precision T value.

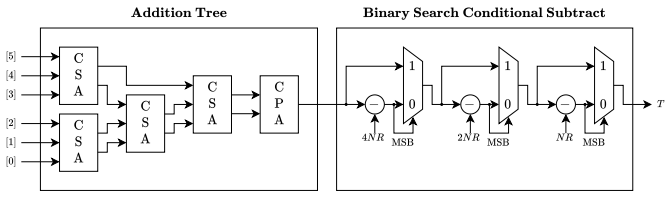

Fig. 2. Modular accumulation tree overview

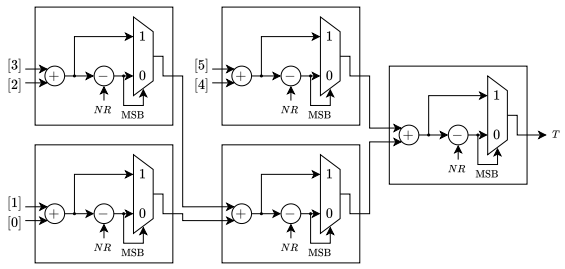

Fig. 3. Modular accumulation binary tree overview

Stage 3: q-computation. Because only lowest w-bits are needed to compute $q = (T \mod R)\mu \mod R$, we *decouple* q computation from the full-precision T accumulation. A cheap word-sized copy of the accumulation executes in parallel to Stage 2 and completes earlier. Hence, the value q is available when the full-precision T accumulation is complete, completely hiding the q computation latency. We avoid any explicit modular arithmetic in this path as everything is already handled implicitly modulo $R = 2^w$. We finally compute qN with a $w \times (\lceil \log_2(N+1) \rceil)$-bit multiplier such that qN is available as T computation finishes.

Stage 4: Final Montgomery reduction. With both T and qN available, we compute $Z = (T + qN)/R$ using an adder, and finally a right shift by w. A final conditional subtraction brings the result into $[0, N)$.

C. Limb Decomposition Strategy

We optimize our implementation to use FPGA-specific resources, such as DSP48E2 blocks, on Virtex Ultrascale FPGAs. DSP48E2 supports 27×18-bit signed or 26×17-bit unsigned multiplications. Standard tiling methods partition large multiplications into DSP blocks. We further reduce DSP usage by using Karatsuba multiplication for the large-integer multiplier before the reduction. We chose a 17-bit word size for all multiplications by the small values, as they fit within the smaller port of the DSP block.

D. Reducing the Depth of Modular Accumulation

The modular accumulation tree shown in Fig. 1 dominates the latency of the overall module. By reducing its latency, we can directly reduce the latency of our modular reduction circuit. We have implemented a CSA tree with binary search conditional subtraction for the $n-1$ multiples of NR in T, as shown in Fig. 2. Two other approaches were explored.

First, we used naive modular adders for each addition node. Having separate modular adders enabled simpler pipelining as shown in Fig. 3, but at the cost of increased clock-cycle latency. Second, we implemented a separate adder and subtraction approach. This reduced clock-cycle latency but required adding pipeline registers to break long carry chains. The most performant and chosen configuration uses CSA and binary search division for the conditional subtractions. The CSA tree maps well onto the FPGA logic and breaks the carry-resolution dependency of the initial approach.

TABLE I
NUMBER OF SUBTRACTIONS INFERRED FOR BLS12-377 AND BLS12-381

Prime	Naive # of Subtractions	Depth of Subtraction Circuit	Inferred # of Subtractions	Depth of Subtraction Circuit
BLS12-377	22	5	13	4
BLS12-381	22	5	12	4

The depth of the conditional subtraction circuit can be further reduced by inferring the maximum possible multiples of NR in T. We can get the maximum conditional subtraction value by computing $\tau_{\max} = \lfloor \sum \frac{\rho_i}{N} + 1 \rfloor, i \in [1, n-1]$. This optimization further reduces the modular accumulation tree latency by one cycle for BLS12-377/381, as shown in Table I.

E. EDA Tooling

The entire design is parameterized by two user-supplied values: N and w. Our scripts derive all dependent parameters, including n, ρ_i, μ, and $\tau_{\max}$, to generate the corresponding RTL configuration. All required number-theoretic computations are implemented in a self-contained TCL library and require no external dependencies beyond AMD Vivado. This automation supports rapid design-space exploration and improves reproducibility, since a new prime field is supported by changing the input parameters while all derived constants are generated automatically from a consistent parameter set.

IV. RESULTS

We evaluate our low-latency modular multiplier for different MSM parameter sets, like BLS12-377/381 prime fields. We synthesized, placed, and routed the design using Vivado 2022.2 for an AMD Alveo U250 FPGA and verified its correctness on hardware. We report implementation results in terms of pipeline stages, LUTs, FFs, and DSPs for different frequency targets. These results are then used to compare the proposed modular multiplier to prior FPGA-based works in terms of area, latency, and overall MSM performance. All compared accelerators target the same Virtex UltraScale+ fabric.

A. Area Consumption and Module Breakdown

Table III summarizes the implementation results of our modular multiplier for the BLS12-377/381 prime fields at 250 and 400 MHz. We report pipeline depth, LUTs, FFs, and DSPs, and further break the design into integer multiplication and modular reduction. The proposed design reaches 17 pipeline stages at 250 MHz and 30 stages at 400 MHz. As expected, the higher frequency target increases LUT and FF usage due to the additional pipeline registers. The results also show that the modular reduction dominates the overall resource consumption. This aligns with the focus of our work, since the main optimization and contribution target the reduction stage within Montgomery multipliers.

TABLE II
COMPARISON TO OTHER LARGE-PRIME MODULAR MULTIPLIERS.

Work	Component	Prime	Frequency	Stages	LUT	FF	DSP
[14]	Full MSM accelerator	BLS12-377	250 MHz	96 (1.85×)	525298 (0.76×)	661146 (1.04×)	2277 (0.55×)
	Single modular multiplier	BLS12-377	250 MHz	39 (2.29×)	43144 (0.68×)	46866 (1.08×)	324 (0.55×)
[17]	Full MSM accelerator	BLS12-377	250 MHz	n.a.	849312 (1.23×)	946253 (1.49×)	9011 (2.2×)
	Single modular multiplier	BLS12-377	250 MHz	22 (1.29×)	24000 (0.38×)	n.a.	413 (0.7×)
Our	Full MSM accelerator	BLS12-377	250 MHz	52	692290	636037	4097
	Single modular multiplier	BLS12-377	250 MHz	17	63773	43279	584

TABLE III
OUR IMPLEMENTATION RESULTS FOR U250 FPGA.

Module	Prime	Frequency	Stages	LUT	FF	DSP
ModMul	BLS12-377	250 MHz	17	63773	43279	584
⌊ IntMul			6	20046	6579	216
⌊ ModRed			11	43727	36700	368
⌊ ModAcc			7	13556	12574	0
ModMul	BLS12-377	400 MHz	30	66919	70876	584
⌊ IntMul			12	21015	13391	216
⌊ ModRed			18	45904	56731	368
⌊ ModAcc			9	14854	15517	0
ModMul	BLS12-381	250 MHz	17	67200	44673	584
⌊ IntMul			6	20394	7445	216
⌊ ModRed			11	46806	37228	368
⌊ ModAcc			7	16587	12716	0
ModMul	BLS12-381	400 MHz	30	66125	71618	584
⌊ IntMul			12	20465	13604	216
⌊ ModRed			18	45660	57252	368
⌊ ModAcc			9	14768	15667	0

Due to Vivado's synthesis behaviour, the reported area consumption of submodules may not add up to the overall module size.

TABLE IV
LATENCY COMPARISON OF FULL MSM WORKLOADS IN MILLISECONDS.

Work	MSM Size	Latency of Phases in MSM (in ms)				
		BC	BA	Speedup	Total [ms]	Speedup
[14]	2^{22}	214	604	(1.85×)	818	(1.51×)
	2^{23}	529	604	(1.85×)	1133	(1.32×)
	2^{24}	1157	604	(1.85×)	1761	(1.19×)
	2^{25}	2412	604	(1.85×)	3016	(1.10×)
	2^{26}	5052	604	(1.85×)	5656	(1.05×)
Our	2^{22}	214	327	(1.00×)	541	(1.00×)
	2^{23}	529	327	(1.00×)	856	(1.00×)
	2^{24}	1157	327	(1.00×)	1484	(1.00×)
	2^{25}	2412	327	(1.00×)	2739	(1.00×)
	2^{26}	5052	327	(1.00×)	5379	(1.00×)

Note: RA is performed in software, not on the FPGA.

B. Area comparison to other MSM accelerators on FPGA

This subsection compares the proposed modular multiplier to prior FPGA-based MSM accelerators. We focus on CycloneMSM [14] and the design by Yrrid and Ingonyama [17], since both provide results for large-prime modular multiplication or full MSM acceleration on FPGA. Table II summarizes the corresponding area and latency results. Overall, our design reduces pipeline latency substantially, but at the cost of higher LUT and DSP usage at the modular-multiplier level.

Comparison to CycloneMSM. The work [14] targets the BLS12-377 curve at 250 MHz on an AMD/Xilinx VU9P FPGA. Their modular multiplier uses 39 pipeline stages, whereas our design requires only 17 stages. This corresponds to a 2.29× lower pipeline latency. The lower latency also reduces register usage slightly, while the LUT and DSP cost increases by 1.48× and 1.8×, respectively. At the full MSM level, CycloneMSM uses a 96-stage point-addition pipeline, whereas our adaptation reduces this to 52 stages at the same clock frequency. This 46% lower latency comes at the cost of 1.32× higher LUT usage and 1.8× higher DSP utilization.

Comparison to Yrrid and Ingonyama. The modular multipliers of Yrrid and Ingonyama [17] use 22 pipeline stages, compared with our 17. This corresponds to a 1.29× lower pipeline latency. The lower latency comes at the cost of 2.66× higher LUT usage and 1.41× higher DSP usage. However, at the full MSM accelerator level, our design is 0.81× and 0.45× smaller in LUT and DSP usage, respectively. A reason for this difference is the underlying coordinate choice. Yrrid and Ingonyama use a batched affine-coordinate approach with Montgomery batch inversion, whereas CycloneMSM and our work use extended coordinates and therefore avoid the overhead of inversion.

C. Results for MSM workloads

Finally, we evaluate the impact of the proposed modular multiplier on end-to-end MSM workloads. Table IV reports the latency of bucket classification (BC), bucket aggregation (BA), and the total MSM computation. For BC, the reduced pipeline latency has only a marginal effect in our evaluation. This is due to two reasons. First, BC scales with the MSM size. Second, the large window size of 16 bits leads to a low probability of bucket conflicts. For smaller window sizes, the conflict probability increases, and the benefit of the lower pipeline latency on BC becomes more pronounced. However, BA shows a different behavior since it is independent of the MSM size and therefore benefits directly from the shorter point-operation pipeline. Compared to CycloneMSM, our design reduces BA latency by 1.85×. As a result, the impact on total MSM latency is strongest for small MSM sizes, where BC contributes less to the overall runtime. In case of size 2^{22} MSMs, our design achieves up to 1.51× lower total latency.

V. DISCUSSION & CONCLUSION

We presented a practical parallel LogJumps-based modular multiplier for FPGA-based MSM acceleration. Our work formalizes the parallel LogJumps reduction, leverages multiplier-internal data-parallel processing, and maps the design to a low-latency hardware architecture. The resulting multiplier design maintains a competitive clock frequency of 400 MHz while reducing modular multiplier latency by up to 2.29× at 250 MHz. This latency reduction allows up to 1.85× and 1.51× more performant bucket aggregation and overall MSM computation, respectively. With these results, we propose new trade-offs for MSM computation and contribute to the performance of ZKP systems.

REFERENCES

[1] S. Atapoor, K. Baghery, H. VL Pereira, and J. Spiessens, "Verifiable fhe via lattice-based snarks," *IACR Communications in Cryptology (CiC)*, vol. 1, no. 1, 2024.

[2] V. Farzaliyev, C. Pärn, H. Saarse, and J. Willemson, "Lattice-based zero-knowledge proofs in action: applications to electronic voting," *Journal of Cryptology*, vol. 38, no. 1, p. 6, 2025.

[3] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 459–474.

[4] T. Z. Foundation, "Zcash digital currency," https://z.cash, Accessed April 2026.

[5] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-Size Commitments to Polynomials and Their Applications," in *Advances in Cryptology - ASIACRYPT 2010*, ser. LNCS, M. Abe, Ed., vol. 6477. Springer, 2010, pp. 177–194. [Online]. Available: https://doi.org/10.1007/978-3-642-17373-8_11

[6] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326.

[7] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Advances in Cryptology – EUROCRYPT 2023*, C. Hazay and M. Stam, Eds. Cham: Springer Nature Switzerland, 2023, pp. 499–530.

[8] C. Liu, H. Zhou, P. Dai, L. Shang, and F. Yang, "Priormsm: An efficient acceleration architecture for multi-scalar multiplication," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 5, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3678006

[9] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symposium on Foundations of Computer Science (SFCS 1976)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1976, pp. 258–263. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SFCS.1976.21

[10] X. Pottier, T. de Ruijter, J. Bertels, W. Legiest, M. Van Beirendonck, and I. Verbauwhede, "Optimsm: Fpga hardware accelerator for zero-knowledge msm," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 2, p. 489–510, Mar. 2025. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/12055

[11] C. F. Xavier, "PipeMSM: Hardware acceleration for multi-scalar multiplication," Cryptology ePrint Archive, Paper 2022/999, 2022. [Online]. Available: https://eprint.iacr.org/2022/999

[12] C. Liu, H. Zhou, L. Yang, J. Xu, P. Dai, and F. Yang, "Gypsophila: A scalable and bandwidth-optimized multi-scalar multiplication architecture," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658259

[13] F. Hirner, F. Krieger, and S. S. Roy, "Chiplet-based techniques for scalable and memory-aware multi-scalar multiplication," *Cryptology ePrint Archive*, 2025.

[14] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "Fpga acceleration of multi-scalar multiplication: Cyclonemsm," *Cryptology ePrint Archive*, 2022.

[15] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao, "Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024, pp. 33–39.

[16] C. Kaya Koc, T. Acar, and B. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[17] Ingonyama, "Deep dive into the latest msm hardware implementation," Online, 2024, https://www.ingonyama.com/post/%20deep-dive-into-the-latest-msm-hardware-implementation, accessed in April 2026.

[18] Y. Domb, "The barrett-montgomery duality," https://hackmd.io/@Ingonyama/Barret-Montgomery, 2025.