MSMAC: Accelerating Multi-Scalar Multiplication for Zero-Knowledge Proof

Pengcheng Qiu, Guiming Wu*, Tingqiang Chu, Changzheng Wei, Runzhou Luo, Ying Yan, Wei Wang, Hui Zhang
Ant Group
Hangzhou, China
{pengcheng.qpc,guiming.wgm,chutingqiang.ctq,changzheng.wcz}@antgroup.com
{luorunzhou.luorunz,fuying.yy,wei.wangwwei,shengchu.zh}@antgroup.com

ABSTRACT

Multi-scalar multiplication (MSM) is the most computation-intensive part in proof generation of Zero-knowledge proof (ZKP). In this paper, we propose MSMAC, an FPGA accelerator for large-scale MSM. MSMAC adopts a specially designed Instruction Set Architecture (ISA) for MSM and optimizes pipelined Point Addition Unit (PAU) with hybrid Karatsuba multiplier. Moreover, a runtime system is proposed to split MSM tasks with the optimal sub-task size and orchestrate execution of Processing Elements (PEs). Experimental results show that MSMAC achieves up to 328× and 1.96× speedups compared to the state-of-the-art implementation on CPU (one core) and GPU, respectively, outperforming the state-of-the-art ASIC accelerator by 1.79×. On 4 FPGAs, MSMAC performs 1,261× faster than a single CPU core.

1 INTRODUCTION

Zero-knowledge proof (ZKP) is a cryptographic protocol that allows the prover to convince the verifier that a computation $y = f(x, \omega)$ is correctly executed, where x is the public input and ω is a secret that only the prover knows. The protocol also guarantees that the secret ω will not be disclosed. ZKP has broad applications such as verifiable outsourcing [10], electronic voting [12], verifiable machine learning [9] and ZK-rollups [1].

ZkSNARK (Zero Knowledge Succinct Non Interactive Argument of Knowledge) is considered one of the most practical ZKP protocols. Its proof is compact and its verification is fast. However, the slow proof generation speed of zkSNARK is a major obstacle to practical application. During the proof generation, some complex operations are required, such as Multi Scalar Multiplication (MSM). The time consumption of MSM can be as high as 70% [11] in proof generation.

MSM is a type of inner product operation defined on elliptic curve, which requires a large amount of point multiplication and point addition on elliptic curve. To reduce proof generation time, accelerators are used to accelerate MSM. Specialized MSM hardware accelerators based on ASIC [11], FPGA [2, 7] and GPU [4, 5] are proposed.

*Corresponding author

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655672

In this article, we propose MSMAC, an FPGA accelerator for large-scale MSM. We apply the Pippenger algorithm to convert the main operations of MSM to point addition, and design a high performance pipelined Point Addition Unit (PAU) for MSMAC. A novel domain-specific Instruction Set Architecture (ISA) is proposed to control the whole MSM process on MSMAC. MSMAC is scalable and can be deployed on multiple FPGAs. Our contributions are summarized as follows:

- We propose a novel MSM accelerator, called MSMAC, which is based on a domain-specific ISA for accelerating MSM. The ISA is designed to simplify hardware implementation, while enabling scalable and high efficient MSMAC architecture.
- We propose a runtime system to efficiently schedule MSMAC's Processing Elements (PEs). Large-sized MSM tasks are split into multiple sub-tasks with optimal sub-task sizes which are searched and kept in advance. Then sub-tasks are dispatched to PEs in a producer-consumer model.
- We propose a pipelined PAU with efficient Montgomery modular multiplier exploiting hybrid multi-way Karatsuba split, which effectively utilizes the most of bits of DSPs in the target FPGA and reduces the DSP resource consumption.
- On a single FPGA, MSMAC achieves a speedup of up to 328× compared to a single CPU core, and a speedup of up to 1.79× and 1.96× compared to state-of-the-art ASIC and GPU implementations, respectively; On 4 FPGAs, the speedup of up to 1,261× can be obtained compared to a single CPU core.

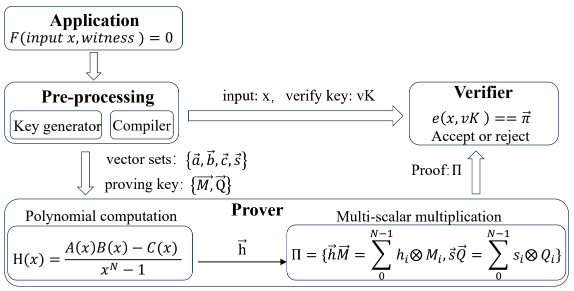
2 BACKGROUND

2.1 The zkSNARK Protocol

The workflow of zkSNARK is shown in Figure 1. Pre-processing randomly generates proving keys and verifying keys for the prover and verifier, respectively. The pre-processing stage also compiles the computation f to be proven as vector sets.

Prover generates a proof based on the vector sets and proving key. The proving keys are elliptic curve (EC) point vectors. The prover computes scalar vector h through polynomial computation, and then performs MSMs. MSM is the inner product between a scalar vector and a point vector, where multiplication is expensive point multiplication on the elliptic curve. The MSM results are essential components of the proof. The verifier can quickly verify its validity by performing a simple calculation.

The size of MSM is related to the computation f to be proven, which may reach billions or even larger. MSM is time-consuming and takes up most of the proof generation time. It is important to accelerate MSM for zkSNARK.



Application
$F(input\ x, witness) = 0$
Pre-processing
Key generator
Compiler
input: x, verify key: vK
Verifier
$e(x, vK) == \vec{\pi}$
Accept or reject
vector sets: $\{\vec{a}, \vec{b}, \vec{c}, \vec{s}\}$
proving key: $\{\vec{M}, \vec{Q}\}$
Proof: Π
Prover
Polynomial computation
$H(x) = \frac{A(x)B(x) - C(x)}{x^N - 1}$
$\vec{h}$
Multi-scalar multiplication
$\Pi = \{\vec{h}\vec{M} = \sum_0^{N-1} h_i \otimes M_i, \vec{s}\vec{Q} = \sum_0^{N-1} s_i \otimes Q_i\}$

Figure 1: zkSNARK protocol

2.2 MSM

MSM is defined as $Q = \sum_{i=0}^{N-1} k_i * P_i$, where k_i is a scalar and P_i is a point on the elliptic curve. MSM requires N expensive point multiplications, with each point multiplication involving multiple point addition and point doubling operations. The Pippenger algorithm is an efficient algorithm that reduces the complexity of MSM. The bit length of the scalar k_i is defined as λ. The Pippenger algorithm splits the scalar k_i into λ/s windows each with the size s, i.e., $k_i = \sum_{j=0}^{\lambda/s-1} 2^{js} m_{ij}$. For window j, the points P_i according to the same $m_{ij}(= l)$ are accumulated into a bucket (B_l). Then the final result is calculated by summing up the weighted bucket B_l in each window.

The Pippenger algorithm is decomposed into three steps:

- Step 1 (bucket accumulation): For each window, the point P_i is accumulated into a bucket according to the s-bit word m_{ij}.

$$B_l = \sum_{i=0}^{N-1} [m_{ij} == l] * P_i, (l = 1, 2, ...2^s - 1) \tag{1}$$

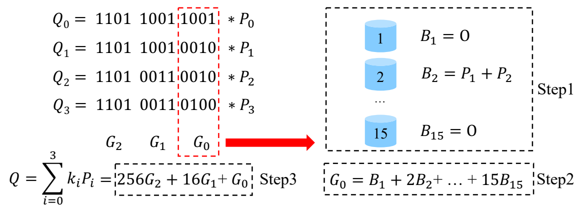
- Step 2 (bucket aggregation) : For each window, accumulate all the buckets to obtain G_j

$$G_j = \sum_{l=1}^{2^s-1} l * B_l \tag{2}$$

- Step 3 (inter-bucket aggregation) : Accumulate G_j to obtain the final result Q

$$Q = \sum_{j=0}^{\lambda/s-1} 2^{js} * G_j \tag{3}$$

An example is shown in Figure 2, where $\lambda = 12$ and $s = 4$. In Step 2, the $2^s - 1$ point multiplications can be converted to $2 * (2^s - 1)$ point additions [4]. Thus, all the operations in Steps 1 and 2 become point additions. Step 3 requires much fewer computations. The aim of our accelerator is to accelerate both Steps 1 and 2.



$Q_0 =$ 1101 1001 1001 $* P_0$
$Q_1 =$ 1101 1001 0010 $* P_1$
$Q_2 =$ 1101 0011 0010 $* P_2$
$Q_3 =$ 1101 0011 0100 $* P_3$
G_2 G_1 G_0
1 $B_1 = 0$
2 $B_2 = P_1 + P_2$
15 $B_{15} = 0$
Step1
$Q = \sum_{i=0}^{3} k_i P_i = 256G_2 + 16G_1 + G_0$
Step3
$G_0 = B_1 + 2B_2 + \ldots + 15B_{15}$
Step2

Figure 2: An example of Pippenger algorithm

2.3 Point Addition

In this paper, we focus on the BN128 curve [3]. The equation for the BN128 curve is $y^2 = x^3 + 3$, where (x, y) is a point in the form of affine coordinates. The point addition operation calculates the sum of two EC points, namely

$$P_3(x_3, y_3) = P_1(x_1, y_1) + P_2(x_2, y_2) \tag{4}$$

EC points can be represented as various coordinates, such as projective coordinates, to simplify point addition operation. The mapping from affine coordinates to projective coordinates can be represented as $(x, y) \longrightarrow (X, Y, Z)$, where $x = X/Z$ and $y = Y/Z$.

Common point addition algorithms cannot handle special cases, e.g., when $P_1 = P_2$ or P_1/P_2 is the zero point on elliptic curve. A complete point addition algorithm [5] can support the point addition on any two EC points. It can avoid point checking and simplify hardware design. We design a pipelined PAU based on the complete point addition algorithm using projective coordinates.

2.4 Related Work

Recently published MSM accelerators include ASIC accelerator [11] and GPU [4, 5]. PipeZK [11] is, to the best of our knowledge, the first ASIC MSM accelerator. It can alleviate load imbalance and be scalable in a coarse-grained manner. However, it only implements Step 1 on hardware and adopts a minor window size which can lead to increased computational overhead when the MSM size is large. GZKP [5] adopts a new parallelization strategy for GPUs, which aggressively combines elliptic curve point operations and exploits fine-grained task parallelism with load balancing for sparse integer distribution. cuZK [4] proposes a new parallel MSM algorithm which is well adapted to the high parallelism provided by GPUs and presents an efficient GPU implementation. Some FPGA based MSM accelerators [7] [2] focus on the BLS12-381 curve. MSMAC targets the BL128 curve, which offers higher performance compared to BLS12-381 curve.

Compared with PipeZK, MSMAC implements both Steps 1 and 2 on FPGA, which results in better end-to-end performance. Additionally, MSMAC supports a larger window size which reduces the computational overhead when dealing with large-sized MSM and achieves a balance between performance and resource utilization. Furthermore, the domain-specific ISA and hardware-software co-design enhances the flexibility and scalability of MSMAC.

3 MSMAC

3.1 Architecture Overview

The overall architecture of MSMAC is shown in Figure 3. MSMAC consists of multiple FPGA cards, each with one Xilinx VP1502 FPGA, one of the Xilinx's newest Versal Premium series FPGAs [8]. Multiple Processing Elements (PEs) can be integrated into one VP1502 FPGA. Each MSMAC PE has one four-stage pipeline, especially designed for accelerating the Pippenger algorithm. This pipeline is similar to a standard processor, but has customized functions defined by our proposed custom ISA.

On the VP1502 FPGA, all PEs, PCIe Controller and DDR4 Controller are connected through the built-in programmable NoC (Network on Chip), which is a high-speed AXI-interconnecting network used for sharing data between all user and built-in IP endpoints. The NoC provides a cross-sectional bandwidth of up to 2.2 Tb/s. Compared to the NoC implemented by the user, the built-in NoC can alleviate routing congestion, and provide system level communication between the various components.

In MSMAC, each PE can handle an independent MSM task. A large-scale MSM task also can be divided into several smaller subtasks, each assigned to one PE. Our runtime system supports a high-efficient task scheduling schemes using automatic task partitioning and a producer-consumer model, detailed in Section 4.

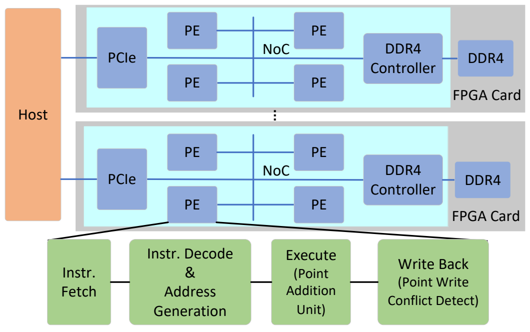

Host
PCIe
PE
PE
PE
PE
NoC
DDR4 Controller
DDR4
FPGA Card
Instr. Fetch
Instr. Decode & Address Generation
Execute (Point Addition Unit)
Write Back (Point Write Conflict Detect)

Figure 3: Overall architecture of MSMAC

3.2 Processing Element (PE)

3.2.1 Microarchitecture. The microarchitecture of each PE is shown in Figure 4. The PE is a domain-specific processor which consists of four stages: fetch, decode, execute and write back. Instead of register files, temporary data is stored in RAMs or FIFOs. Buckets for each window are kept in the Bucket RAM, and intermediate results of Step 2 are stored in the Temp RAM. Each input scalar is encoded into a signed-digit representation by the Scalar Convert module. The signed-digit representation reduces the size of the Bucket RAM by about half [7]. Input and output streams are buffered by the Input and Output FIFOs, respectively. Additionally, there is a Conflict FIFO for buffering conflicting data. The execute unit is a pipelined Point Addition Unit (PAU), which receives points P_1 and P_2 every cycle, and outputs the result P_3.

The instructions are stored in the Instruction RAM and will be fetched to the decoder. The *padd* instruction is proposed to perform point addition, and is executed as follows:

(1) The decoder parses the *padd* instruction and generates control signals for RAMs, FIFOs, and multiplexers.
(2) Points P_1 and P_2 are read out from RAMs or FIFOs and sent to PAU.
(3) PAU performs point addition on P_1 and P_2.
(4) The result P_3 is stored into one of the three destinations (Bucket RAM, Temp RAM or Output FIFO) according to the corresponding write address.

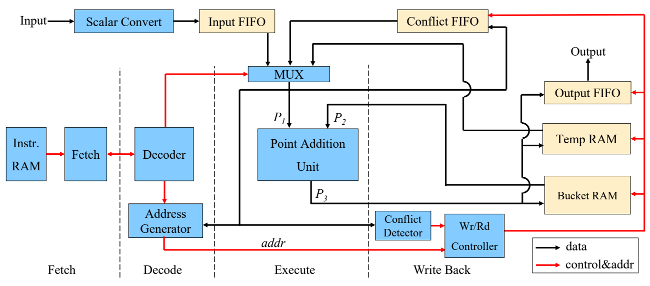


Figure 4: PE microarchitecture

The point addition result is obtained after L_{PAU} cycles, where L_{PAU} is the stage number of the PAU pipeline. During Step 1, if the bucket B_l is in process and a new point to be accumulated to B_l arrives, a conflict occurs. To address this conflict, we design a conflict detector. After a conflict is detected, the result of PAU is discarded, and the point and corresponding s-bit scalar word is written to the Conflict FIFO. The data in Conflict FIFO will be processed when the MSM input stream ends or the number of data in Conflict FIFO exceeds a pre-defined threshold. However, the impact of conflicts on PE performance is insignificant as the probability of conflicts is much low.

To make full use of the pipelined PAU, PE processes λ/s windows in a batch, i.e., point P_i is accumulated to λ/s buckets according to the scalar k_i under one *padd* instruction. Since each window has its own buckets, batch processing can reduce the probability of conflicts by λ/s times further. Batch processing can also reduce the memory bandwidth requirement as input points and scalars only need to be read once for all the calculations they are involved in.

3.2.2 ISA. We design a specialized ISA for MSM with four instructions as shown in Table 1. *padd* is the only arithmetic instruction. The *jmp* and *set* instructions are used to control MSM workflow. *nop* will be inserted to resolve data dependency when necessary. The format of the *padd* instruction is shown in Figure 5. One *padd* instruction can perform a batch of point additions, and the batch size is decided by the *size* field of the instruction. In batch processing, the control information such as the *src_sel* is reused, but the address is automatically updated according to the base address and step fields. The step field means the stride length for address increase.

Normally, a *jmp* instruction may come with overhead even with complicated branch prediction techniques. We adopt a trick to eliminate the effect of conditional jump, ensuring the efficiency of our ISA. A batched *padd* instruction takes several cycles. The *jmp* after *padd* can be fetched and executed during the *padd* is running. The next instruction can also be fetched before the *padd* is done. In this way, conditional jump comes with no overhead.

In Step 1, performing a conditional jump in advance does not affect the final outcome. Let's take a partial workflow of Step 1 as an example, which is illustrated in Figure 6. After executing

Table 1: Our proposed ISA

Instruction	Description
padd	Point addition
jmp	Conditional jump
set	Loop counter setting
nop	No operation

jmp0, MSMAC may jump to *padd0* (or *padd1*) to process data in the Input FIFO (or the Conflict FIFO). Performing *jmp0* in advance may change the execution order of *padd0* and *padd1* since the status of Conflict FIFO may be changed at the last cycle of batch processing. Nevertheless, the Conflict FIFO has sufficient buffer space to accommodate the data after almost-full warning. Delaying the execution of *padd1* slightly does not lead to Conflict FIFO overflow and result errors.

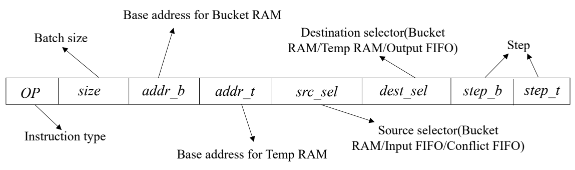

Base address for Bucket RAM
Batch size
Destination selector(Bucket RAM/Temp RAM/Output FIFO)
Step
OP
size
addr_b
addr_t
src_sel
dest_sel
step_b
step_t
Instruction type
Base address for Temp RAM
Source selector(Bucket RAM/Input FIFO/Conflict FIFO)

Figure 5: *padd* instruction

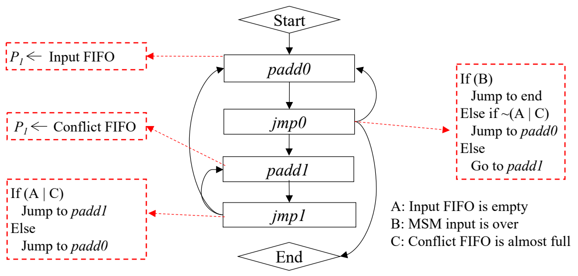


Figure 6: Partial workflow of Step 1

3.2.3 *PAU.* We design a pipelined PAU based on a complete point addition algorithm [6], and its data flow diagram is shown in Figure 7. The kernel of PAU is a modular multiplier. Our modular multiplier utilizes Montgomery multiplication which requires three large integer multiplication, as shown in Algorithm 1. For our target BN128 curve, the three 256-bit multiplications in Montgomery multiplication have been optimized and tailored for the target FPGA.

Suppose a and b are n-bit integers, where $a = a_1 * 2^{n/2} + a_0$, $b = b_1 * 2^{n/2} + b_0$. The schoolbook method requires four $n/2$-bit multipliers. The 2-way Karatsuba split only requires three $n/2$-bit multipliers, as shown below:

$$a * b = 2^{2n} a_1 b_1 + 2^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + a_0 b_0 \quad (5)$$

If the 2-way Karatsuba split is applied recursively, a 256-bit multiplier can be implemented with 81 multipliers (DSPs) with widths ranging from 16 to 18. However, the DSPs of the VP1502 FPGA are 27 × 24 multipliers, meaning that the most significant bits in DSPs is wasted if this method is used. To fully utilize DSP resources, we propose a hybrid Karatsubasua split scheme, as shown in Figure 8. Through this scheme, only 54 DSPs are required in total.

Algorithm 1: Montgomery Modular Multiplication

```
Input  : $X, Y, R, M; X, Y \in [0, M-1]; gcd(R, M) = 1$
Output : $Z = XYR^{-1} mod M$
1 $t = XY; m = t(-M^{-1}) mod R; Z = (t + mM)/R$
2 if $Z \geq M$ then $Z = Z - M$
```

In Algorithm 1, the second 256-bit multiplier only requires the lower half of the result, and can be divided into 5 L1 multipliers. 2 of the 5 L1 multipliers can be further optimized. As a result, 43 DSPs are required in total. Similar optimization can be applied to the third 256-bit multiplier. Moreover, we can replace some DSPs with LUTs to balance resource utilization.

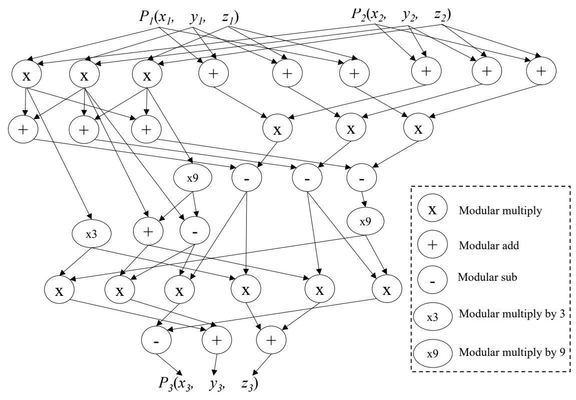


Figure 7: Point Addition Unit

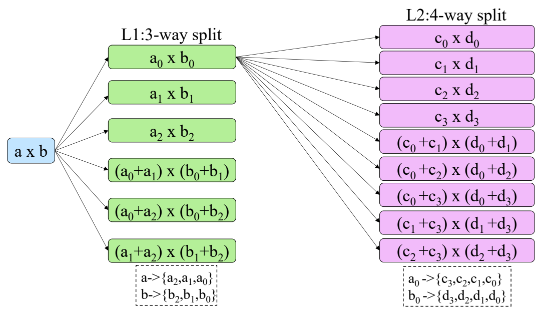

a x b
L1:3-way split
$a_0 \times b_0$
$a_1 \times b_1$
$a_2 \times b_2$
$(a_0+a_1) \times (b_0+b_1)$
$(a_0+a_2) \times (b_0+b_2)$
$(a_1+a_2) \times (b_1+b_2)$
a->{a_2,a_1,a_0}
b->{b_2,b_1,b_0}
L2:4-way split
$c_0 \times d_0$
$c_1 \times d_1$
$c_2 \times d_2$
$c_3 \times d_3$
$(c_0+c_1) \times (d_0+d_1)$
$(c_0+c_2) \times (d_0+d_2)$
$(c_0+c_3) \times (d_0+d_3)$
$(c_1+c_3) \times (d_1+d_3)$
$(c_2+c_3) \times (d_2+d_3)$
a_0->{c_3,c_2,c_1,c_0}
b_0->{d_3,d_2,d_1,d_0}

Figure 8: Hybrid Karatsuba split

3.3 Performance Model

The Pippenger algorithm has an important parameter s (i.e., window size). When designing MSMAC, s is constrained by memory resources required by Bucket RAM. The number of PE is also limited by available FPGA resources, especially the DSP resources. One VP1502 FPGA can accommodate four PEs with $s \leq 13$.

The total computation of Steps 1 and 2 is

$$Com_{total} \approx (\lambda/s) * N + 2 * (\lambda/s)(2^s - 1). \tag{6}$$

The optimal s to minimize Com_{total} under different N are shown in Figure 9. When $N \geq 2^{17}$, the optimal s is 13. As N gradually decreases from 2^{17}, the optimal s decreases accordingly. On MSMAC, s can be adjusted for smaller N to achieve optimal performance.

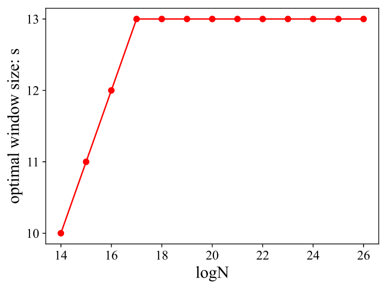


Figure 9: Optimal window size under different N

4 RUNTIME SYSTEM

MSMAC runtime system is proposed to monitor and orchestrate PE execution for MSMAC. Our MSMAC runtime system can efficiently schedule tasks onto the PEs through applying a producer-consumer model.

The runtime system is shown in Figure 10. It consists of runtime user space and kernel space, acting as producer and consumer, respectively. The producer is responsible for dividing large-scale MSM tasks from multiple processes/threads into sub-tasks. Each sub-task is assigned an ID based on the corresponding MSM task it belongs to. These sub-tasks are created and sent to the sequencer of the consumer. The producer and consumer execute asynchronously to release the capacity of all PEs on FPGAs.

In the consumer, the sequencer receives sub-tasks from the producer and dispatches them to worker queues with available slots using a round-robin strategy. A worker dispatcher and a worker queue are created for each PE. The worker dispatcher is responsible for dispatching sub-tasks in the worker queue to PE. The next sub-task can be dispatched while current sub-task is running since there is a ping-pong buffer in on-board DDR for each PE, overlapping the data transfer with computation.

5 EXPERIMENTAL RESULTS

5.1 Implementation

Our experiments are conducted on a server with two 2.60 GHz Intel Xeon Platinum 8385P CPUs (each with 64 logical cores) and 376 GB DRAM. It is equipped with four FPGA cards, each with one VP1502 FPGA. We compare MSMAC with CPU implementation and the state-of-the-art solutions. We ran Halo2 [1], a well-known library for zero-knowledge proof, on the same CPU.

Our MSMAC with four PEs on one FPGA can run at 250MHz, and its resource utilization is shown in Table 2. The DSP utilization has reached 87% which becomes the bottleneck to implement more PEs.

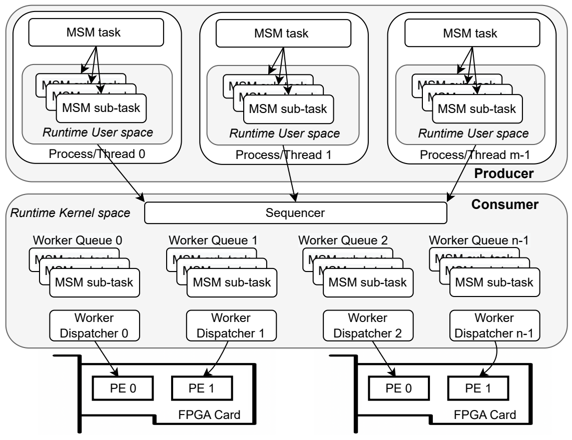


Figure 10: Runtime system

Table 2: Resource utilization

	DSP	LUT	BRAM	URAM
MSMAC	6,480	1,290k	330	768
Available	7,440	1,720k	2,541	1,301
Utilization	87%	75%	13%	59%

5.2 Performance

We measured the performance of MSMAC on 1, 2 and 4 FPGAs. The latency includes the data transfer time between the host and FPGAs and the running time of both FPGAs and host. As mentioned in Section 4, an MSM task is split into smaller sub-tasks which are sent to available PEs on FPGAs. The results of PEs are collected and processed by the host to obtain the final result.

The performance of MSMAC for different sub-task sizes on various amounts of FPGAs when $N = 2^{20}$ and $N = 2^{25}$ is shown in Figure 11. The sub-task size is denoted as k. It can be seen that the optimal sub-task size increases as N increases, and decreases as the number of FPGAs increases. A smaller sub-task size results in more sub-tasks, which is beneficial to overlap data transfer with computation. However, if there are too many sub-tasks, the increased scheduling and aggregation overhead may outweigh the benefit of overlapping. Therefore, there is an optimal sub-task size to balance data transfer, scheduling and computation. When the number of FPGAs increases, optimal sub-task size decreases since more PEs are available and more sub-tasks are needed.

Thus, we performed an exhaustive search to find the optimal sub-task sizes for different N and various amounts of FPGAs in advance. The optimal sizes are then kept in the runtime system and used to split actual MSM tasks according to N and available FPGAs.

5.3 Performance Comparison

The performance comparison between MSMAC and CPU is shown in Table 3. Compared to a single core, one FPGA achieves a maximum speedup of 328×, while four FPGAs achieve a maximum speedup of 1,261×.

Table 3: Performance comparison between MSMAC and CPU

Size	CPU		MSMAC		Speedup (1 FPGA)		Speedup (4 FPGAs)	
	1 core	64 cores	1 FPGA	4 FPGAs	v.s. CPU (1 core)	v.s. CPU (64 cores)	v.s. CPU 1 core)	v.s. CPU (64 cores)
2^{18}	2.45s	122.47ms	10.65ms	6.30ms	230×	11×	389×	19×
2^{19}	4.63s	218.28ms	19.45ms	9.79ms	238×	11×	473×	22×
2^{20}	9.09s	399.61ms	34.15ms	14.39ms	266×	12×	632×	28×
2^{21}	16.72s	681.36ms	58.87ms	22.56ms	284×	12×	738×	30×
2^{22}	31.25s	1.13s	114.48ms	31.43ms	273×	10×	994×	36×
2^{23}	62.44s	1.39s	200.65ms	54.49ms	311×	7×	1,146×	26×
2^{24}	117.01s	2.60s	373.77ms	101.02ms	313×	7×	1,158×	26×
2^{25}	230.58s	5.04s	736.68ms	192.34ms	313×	7×	1,199×	26×
2^{26}	465.71s	10.16s	1.42s	369.30ms	328×	7×	1,261×	28×

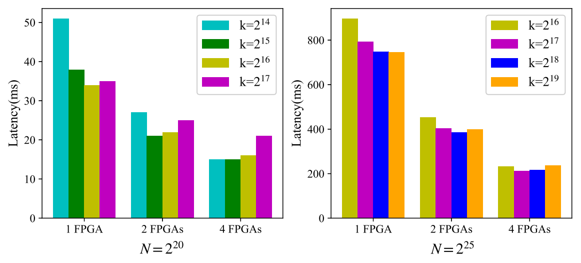


Figure 11: MSMAC performance for different sub-task sizes when $N = 2^{20}$ and $N = 2^{25}$

Table 4: Performance comparison between MSMAC and other accelerators

Size	PipeZK [11]	GZKP [5]	MSMAC (1 FPGA)	MSMAC v.s. PipeZK	MSMAC v.s. GZKP
2^{18}	16ms	15ms	10.65ms	1.50×	1.41×
2^{19}	32ms	-	19.45ms	1.65×	-
2^{20}	61ms	45ms	34.15ms	1.79×	1.32×
2^{22}	-	0.17s	114.48ms	-	1.48×
2^{24}	-	0.72s	373.77ms	-	1.93×
2^{26}	-	2.79s	1.42s	-	1.96×

We also compare MSMAC with other MSM accelerators in the literature, as shown in Table 4. MSMAC can achieve a maximum speedup of 1.79× compared to PipeZK (a state-of-the-art ASIC accelerator). Compared to GZKP (a state-of-the-art GPU implementation), MSMAC can achieve a maximum speedup of 1.96×. GZKP requires additional preprocessing before GPU processing. The reported time does not include the time of preprocessing. Furthermore, the time of data transfers is not included.

PipeZK is an ASIC accelerator and has four PEs for BN128 curve. MSMAC adopts more optimized parameters and proposes a flexible ISA. MSMAC can choose optimal window size and sub-task size according to MSM size and available FPGAs. While PipeZK accelerates only Step 1, MSMAC accelerates both Steps 1 and 2. Furthermore, MSMAC processes the λ/s windows in a batch to improve the utilization of pipelined PAU. These advantages of our FPGA implementation result in higher performance than PipeZK.

6 CONCLUSION

The time-consuming proof generation is a major challenge for ZKP applications. This paper proposes MSMAC, an FPGA-based accelerator to accelerate MSM, a kernel in ZKP. MSMAC adopts a specially designed ISA and can be deployed on multiple FPGAs. It is efficiently scheduled by the runtime system. Compared to a single core, MSMAC on one FPGA achieves a speedup of up to 328×. MSMAC is up to 1.96× faster than the state-of-the-art accelerators.

REFERENCES

[1] 2023. ZERO-KNOWLEDGE ROLLUPS. https://ethereum.org/en/developers/docs/scaling/zk-rollups/
[2] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. 2022. Fpga acceleration of multi-scalar multiplication: Cyclonemsm. *Cryptology ePrint Archive* (2022).
[3] Paulo SLM Barreto and Michael Naehrig. 2005. Pairing-friendly elliptic curves of prime order. In *International workshop on selected areas in cryptography*. Springer, 319–331.
[4] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. 2022. Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *Cryptology ePrint Archive* (2022).
[5] 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*. 340–353.
[6] Joost Renes, Craig Costello, and Lejla Batina. 2016. Complete addition formulas for prime order elliptic curves. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*. Springer, 403–428.
[7] Charles F Xavier. 2022. Pipemsm: Hardware acceleration for multi-scalar multiplication. *Cryptology ePrint Archive* (2022).
[8] Xilinx. 2022. *Versal Premium ACAPs: Breakthrough Integration of Networked IP on a Power-Optimized, Adaptable Platform*. https://docs.xilinx.com/v/u/en-US/wp519-versal-premium-intro
[9] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2039–2053.
[10] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 863–880.
[11] 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 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).
[12] Zhichao Zhao and T-H Hubert Chan. 2016. How to vote privately using bitcoin. In *Information and Communications Security: 17th International Conference, ICICS 2015, Beijing, China, December 9–11, 2015, Revised Selected Papers 17*. Springer, 82–96.