ReZK: A Highly Reconfigurable Accelerator for Zero-Knowledge Proof

Hao Zhou, Changxu Liu, Lan Yang, Li Shang, *Member, IEEE*, and Fan Yang, *Member, IEEE*

***Abstract*— Zero-knowledge proof (ZKP) plays a significant role in privacy protection technology. However, the proof generation phase requires considerable time and hardware resources. In this phase, Number Theoretic Transform or Inverse Number Theoretic Transform (NTT/INTT) in polynomial computation, as well as Multiple Scalar Multiplication (MSM), are bottlenecks that dominate the execution time. In this paper, we propose a highly reconfigurable accelerator ReZK to accelerate ZKP proof generation phase, focusing on NTT/INTT and MSM. According to the configurations, ReZK can be configured as NTT, INTT, and MSM with variable sizes and bit-widths by adjusting the data path between on-chip memories and arithmetic cores. As the basic unit of arithmetic cores, the reconfigurable processing element (PE) in ReZK is composed of pipelined modular multipliers and modular adders that support variable bit-widths. It can perform butterfly or arithmetic operations. Based on the reconfigurable PEs, the ReZK core can implement NTT/INTT with different sizes and bit-widths, or a fully pipelined point adder (PADD). Additionally, we propose a modularized MSM scheduling architecture to support various bit-widths. The on-chip memories are also well organized for reuse. In NTT/INTT mode, 4-way 256-bit or 2-way 384-bit NTT/INTT can be computed in parallel. In MSM mode, for different elliptic curves, ReZK is capable of processing 4-way 256-bit or 2-way 384-bit MSM in parallel.**

***Index Terms*— Zero-knowledge proof, variable bit-width, parallel computation, number theoretic transform, multi-scalar multiplication.**

I. Introduction

ZERO-KNOWLEDGE proof (ZKP) functions as a protocol enabling a prover to substantiate a statement to a verifier without any information leakage. As an important privacy protection technology, ZKP has been applied in multiple scenarios. In the context of blockchain evolution, ZKP has demonstrated its paramount significance in specific scenarios such as blockchain scaling [1], [2], decentralized storage [3], smart contracts [4], and decentralized identity authentication systems [5]. Through ZKP, multi-party computation and federated learning can also fully leverage the inherent value of safeguarded data. Apparently, the value of ZKP will continue to grow with increasing privacy protection requirements.

However, the generation of proof remains a time-consuming process. When compared to the verification phase, the proof generation phase is considerably longer, potentially taking several minutes for a single transaction [6]. This bottleneck significantly hampers the broader adoption of ZKP and its integration into various applications. Therefore, many studies on hardware acceleration for proof generation have been proposed, such as [7], [8], [9], [10], [11], [12], [13], and [14]. Among these, the primary focus is on NTT/INTT and MSM, which dominate the proof generation phase, with MSM alone accounting for at least 70% of the computation time [11].

Additionally, in real-world scenarios, the bit-widths of NTT/INTT and MSM can vary from 256-bit to 384-bit, or even 768-bit. This is mainly related to the elliptic curves (ECs). Table I shows the commonly used ECs in ZKP and corresponding bit-widths of NTT/INTT and MSM. There are several bit-widths required for different ZKP systems that based on various ECs. Therefore, supporting variable bit-widths will significantly improve the flexibility of the accelerator and broaden its application scenarios.

TABLE I
Commonly Used Elliptic Curves in ZKP and Corresponding Bit-Widths of NTT/INTT and MSM

ECs	BN254	BLS12-377	BLS12-381
NTT/INTT	254-bit	253-bit	255-bit
MSM	254-bit	377-bit	381-bit

Meanwhile, in some applications, different ZKP systems with various circuit sizes or protocols would be integrated to work collaboratively. For example, in Zcash [6], [15], the circuit size differences between *Sprout*, *Sapling_spend*, and *Sapling_output* may vary by several times or even dozens of times. Similarly, within the zkSaaS framework [16], there are workloads that simultaneously generating proofs for different ZKP systems, such as Groth16 [17] and Plonk [18], as the framework provides proof generation services for multiple users. However, as shown in Table II, the unbalanced workloads that may affect application efficiency will be introduced due to different circuit sizes, protocols between various ZKP systems, and computation complexity between

Received 7 May 2024; revised 20 August 2024; accepted 17 September 2024. This work was supported in part by the National Key R&D Program of China under Grant 2023YFB2704600 and in part by the National Natural Science Foundation of China (NSFC) Research Projects under Grant 92373207 and Grant 62090025. This article was recommended by Associate Editor J. Zhang. *(Corresponding author: Fan Yang.)*

Hao Zhou, Changxu Liu, Lan Yang, and Fan Yang are with the State Key Laboratory of Integrated Chips and Systems and the School of Microelectronics, Fudan University, Shanghai 200433, China (e-mail: yangfan@fudan.edu.cn).

Li Shang is with the State Key Laboratory of Integrated Chips and Systems and the School of Computer Science, Fudan University, Shanghai 200433, China.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2024.3468033.

Digital Object Identifier 10.1109/TCSI.2024.3468033

TABLE II

ANALYZING ABOUT DIFFERENT ZKP WORKLOADS [11], [19]

Protocol	MSM Proportion	Polynomial Proportion	Proof Size	MiMC Prover Time	SHA-256 Prover Time
Groth16 [17]	70-80%	20-30%	0.2 kB	16.5s	1.4s
Plonk [18]	85-90%	10-15%	0.5-1 kB	5.6s	6.6s

† The prover time and proportion of MSM and Polynomial will vary depending on the circuit size and other parameters. The circuit size can vary from 2^{13} to 2^{26} in various applications.

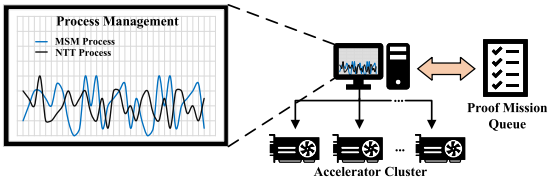


Fig. 1. Unified accelerator cluster.

NTT/INTT and MSM. Therefore, as shown in Fig. 1, multiple unified accelerators which can be dynamically configured and reused as NTT/INTT or MSM will be flexibly scheduled and deployed to process various workloads and help improve the application efficiency.

Existing works are mainly focused on fixed bit-width and separated NTT/INTT/MSM modules. PipeZK [7] proposes an end-to-end pipelined ASIC-based accelerator design, featuring two subsystems for NTT and MSM acceleration. To process different ECs with various bit-widths, PipeZK separately tailors corresponding NTTs and MSMs. SZKP [20] presents a scalable accelerator framework. Using high-level synthesis (HLS) design, SZKP conducts a detailed exploration of the ZKP accelerator design space. However, faced with different ECs, there is almost no flexibility for both PipeZK and SZKP after tape-out. To address the evolving requirements, ZPU [8] has been developed as a more versatile processing unit. ZPU is grounded in an Instruction Set Architecture (ISA) to enhance programmability. Operations such as polynomial computation, MSM, or other relevant tasks can be executed through the corresponding instruction group. However, the data transfer between Processing Elements (PE) consumes multiple cycles. The bit-width of PE unit is configured as 256-bit, presenting challenges in performing multiplications on larger bit-width elliptic curves. Additionally, when handling operations with smaller bit-widths, ZPU may inefficiently utilize redundant resources.

SAM [9], BSTMSM [10], PipeMSM [11], and CycloneMSM [12] propose individual NTT and MSM acceleration units designed for FPGA. In comparison to ASIC designs, these works offer enhanced flexibility. However, the performances of implementations on FPGA platform are relatively lower than those on ASIC frequency. Apart from that, GZKP [13] and cuZK [14] both use GPU accelerate ZKP, particularly for MSM. Due to higher frequency and remarkable resources, the GPU platform commonly yields excellent performance and programmability but also has nonnegligible cost and power.

To address the different bit-width needs of the ZKP system and improve the utilization of resources, we propose a highly Reconfigurable ASIC-based ZKP accelerator (ReZK). While reserving flexibilities to meet the various bit-width requirements, the ReZK efficiently leverages the advantages of the ASIC platform in frequency, cost, and power. Moreover, the ReZK merges NTT and MSM into one accelerator. This reconfigurable accelerator supporting multiple bit-widths and ZKP primitives can greatly improve the flexibility of ASIC accelerators to support various protocols after fabrication. According to configuration, the ReZK can process NTT or MSM computation in different bit-widths.

In summary, our contributions can be summarized as follows.

- We introduce a fully pipelined reconfigurable modular addition supporting multiple bit-widths. Based on reconfigurable modular adder and multiplier, we introduce a fully pipelined reconfigurable PE unit, such as 256-bit and 384-bit. According to the configuration, the PE unit can process butterfly operations or arithmetic operations under different bit-widths.
- With the PE unit, we design a reconfigurable core that can be configured as 2-way 256-bit, or 1-way 384-bit NTT/INTT/PADD datapath. The size of NTT/INTT ranges from 2^6 to 2^{10}, and the PADD is fully pipelined.
- We realize NTT/INTT with configurable size from 2^{12} to 2^{20} by employing two dimensions NTT algorithms. Based on higher dimension recursive algorithm and configuration, larger sizes can be supported. In NTT/INTT mode, 4-way 256-bit or 2-way 384-bit NTT/INTT can be computed in parallel.
- We propose a modularized MSM scheduling method to support various bit-widths in ReZK. Employing and modularized combining the homogeneous scheduling channels, the ReZK can support 4-way 256-bit or 2-way 384-bit MSM in parallel with bucket resources reused in all modes.
- We implement the proposed accelerator in RTL and synthesize it on TSMC 28 nm technology. Under an 800 MHz clock, the area of ReZK is 27.97 mm^2. Among them, 81.4% of resources are reusable for both NTT and MSM. In NTT mode, the ReZK achieves 1.6-29.6× speedups in 256-bit mode, compared with existing implementations. In MSM mode, the ReZK achieves 4.6×-28.6× speedups in 256-bit mode, and 2×-15.6× speedups in 384-bit mode, compared with existing implementations. Compared to NTT/INTT and MSM computation in the overall proving phase based on CPU, ReZK achieves a speedup of over 119×. Compared to the overall proving phase based on CPU, ReZK improves over 3.7×.

The organization of the rest of the paper is as follows. In Section II, the background is reviewed. In Section III, the architecture of ReZK is proposed. In Section IV, we describe the data flow of ReZK in NTT and MSM modes, respectively. Section V shows the implementation results and comparison with other works. In Section VI, we conclude the paper.

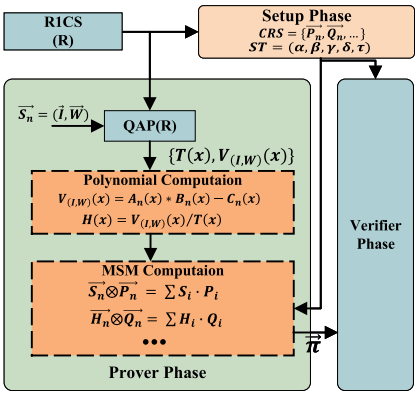


Fig. 2. The workflow of zk-SNARK using the Groth16 protocol as an example.

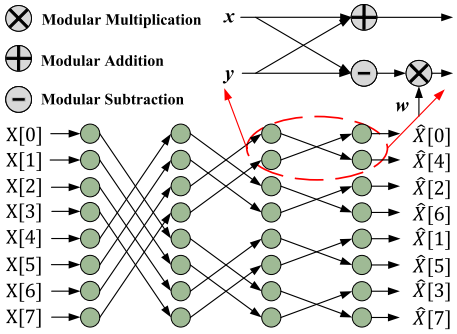


Fig. 3. The example of NTT with size = 8.

II. Background

A. The Zero-Knowledge Proof Protocol

Zero-knowledge proofs allow the prover to convince the verifier of the correctness of a specific statement without divulging any useful information to the verifier. In the zk-SNARK protocol, the proof of a specific statement is converted into the proof of witness W that makes the relevant function true, and the actual ZKP-type problem is therefore converted into a problem that can be solved by computers.

The whole workflow of zk-SNARK, for example, Groth16 [17] shown in Fig. 2, consists of the Setup, Prover, and Verifier phases. Given a Rank-1 Constraint System (R1CS) describing function, the setup phase generates a Common Reference String (CRS) which is also called the prover and verifier key pair. Both the prover and verifier need the CRS to generate or verify proof. At the same time, the R1CS (R) as input is sent to the prover to compute corresponding Quadratic Arithmetic Programs (QAP) with the instance and witness pair (I, W). The outcome of QAP consists of several polynomials like $V_{(I,W)}(x)$ and $T(x)$ called target polynomial. For valid witness, the polynomial $V_{(I,W)}(x)$ is divisible by $T(x)$. Coupling with the computation of $V_{(I,W)}(x) = A_n(x) * B_n(x) - C_n(x)$ in QAP, the computation of $H(x) = V_{(I,W)}(x)/T(x)$ needs a large amount of polynomial computation. Employing CRS generated from the setup phase, the prover sequentially performs the dot product between scalar vectors and point vectors from CRS many times. Moreover, this procedure called MSM computation is resource/time consuming. After that, the proof $\vec{\pi}$ is generated and sent to the verifier.

Among the three stages, the prover phase may take up to several minutes. In this stage, most time is spent on polynomial computation and multi-scalar multiplication, which need to be sped up.

B. Number Theoretic Transform

In polynomial multiplication, the traditional method requires a quadratic number of multiplications, with a time complexity of up to $O(n^2)$. In ZKP, the size of the polynomial will be up to several millions, while the bit-width of polynomial coefficients is several hundreds. The traditional method will cost a huge amount of resources and time. The NTT, $\widehat{X}[i] = \sum_{j=0}^{n-1} \mathrm{X}[j]\omega^{ij} \ mod \ q$, transforms polynomial to evaluation representation. Here ω^{ij}, called twiddle factor, is the n-th primitive root of modulo q and n is the degree of the polynomial. By converting polynomial coefficients $\mathrm{X}[n]$ and $\mathrm{Y}[n]$ to evaluation representation, the pointwise product between two transformed vector $\widehat{X}[i]$ and $\widehat{Y}[i]$ is sequentially processed. Following that, the result vector is converted to polynomial in coefficient form. The time complexity of polynomial multiplication reduces to $O(nlogn)$ through NTT/INTT. The computation resources and time required for polynomial multiplication can be significantly reduced.

The calculation process of INTT, which is described as $\mathrm{X}[j] = n^{-1} \sum_{i=0}^{n-1} \widehat{X}[i]\omega^{-ij} \ mod \ q$, can reuse the hardware structure of NTT apart from that the twiddle factor is substitute as ω^{-ij} and the output is multiplied by n^{-1}. Here, n^{-1} and ω^{-ij} are the multiplicative inverse of polynomial degree n and ω^{ij}.

Fig. 3 shows an example of Decimation-in-frequency (DIF) NTT with $n = 8$. As the basic unit of NTT/INTT, butterfly operation consists of finite field multiplication, addition, and subtraction. The NTT is split into $log2(n)$ stages with each stage having $\frac{n}{2}$ butterfly operations. The stride of i-th stage is $\frac{n}{2^{i+1}}$. After NTT, the output is in reverse order and needs to perform bit reversal operations to transform the output to normal order.

The commonly used pipelined architectures consist of Single-path Delay Feedback (SDF) [21], Multi-path Delay Commutator (MDC) [22], and Single-path Delay Commutator (SDC) [23]. Compared with MDC architecture, the SDF architecture takes only one coefficient per cycle. Therefore, the SDF architecture needs lower bandwidth which is even more important for our design that supports multi-way parallelism. However, when the latency of the butterfly unit is larger than 2 cycles, there are multiple data collisions in the NTT. Therefore, the SDC architecture is employed in this work. As shown in Fig. 4, each stage of NTT employs one butterfly unit. For storing $\frac{n}{2^{i+1}}$ coefficients in the i-th stage, two on-chip memories are needed in each stage. In each stage, the first half of input coefficients are stored in memory. Input coefficient and coefficient buffered in memory are sent to butterfly unit when the on-chip memory is full. The output of one branch of the butterfly unit is sent directly to the next stage, and the output of the other branch is cached in memory. It is also sent to the next stage when memory is full. By bypassing the first few units, the degree supported by SDC architecture is variable.

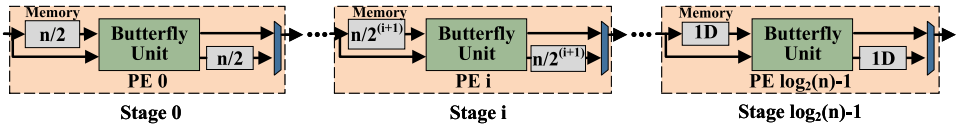


Fig. 4. The single-path delay commutator NTT/INTT architecture.

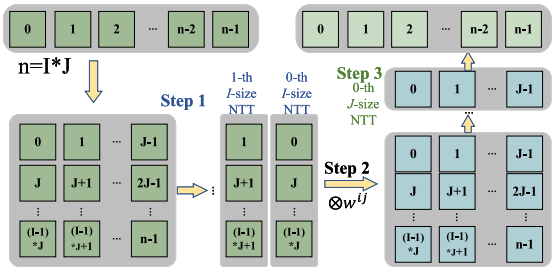


Fig. 5. The algorithm flow of 2D-NTT.

However, though employing SDC architecture, finite hardware resources are not enough to meet very large polynomial orders which may be up to 2^{26}. In [24], two-dimensional NTT (2D-NTT) is introduced to substitute a large degree one-dimensional NTT/INTT (1D-NTT/INTT). As shown in Fig. 5, the flow of 2D-NTT is described. 1D-NTT with $n = I \times J$ is divided into I coefficient vectors of J size and combined into a 2D matrix. The 2D-NTT is converted to multiple independent 1D-NTTs which can be processed in parallel. Multiple I-size 1D-NTTs on columns are first processed. Before being restored to the matrix in normal order, the twiddle factors are multiplicated to the output of the first step. In the third step, multiple J-size 1D-NTTs on rows are processed and that 2D matrix is reshaped to one dimensional matrix.

C. Multi-Scalar Multiplication

MSM, as another important computation in ZKP, takes the most expensive resources and time. It directly affects the efficiency of proof generation. Performing dot-product between n-size scalar and point vectors, as described in $\overrightarrow{S_n} \bigotimes \overrightarrow{P_n} = \sum_{i=0}^{n-1} s_i \cdot p_i$, MSM employs a huge amount of point multiplication (PMULT) represented as $[\cdot]$. The PMULT here requires several times point double (PDBL) and point addition (PADD) based on elliptic curves (EC) $y^2 = x^3 + ax + b$. The bit-width of points on EC is up to several hundreds. Moreover, the PDBL and PADD entail a lot of modular arithmetic such as modular multiplication and modular addition in large finite field. In ZKP, the size of MSM can be up to several millions. There is no doubt that MSM will consume a quite large cost.

A variety of work has been proposed to speed up MSM. Among them, the most popularized algorithm is Pippenger algorithm [25] (or so called bucket method). In direct MSM computation, each PMULT of scalar and point pair needs repeat PDBL and PADD operations. The core idea of the bucket method is to eliminate repeat PDBL and PMULT by first dividing the scalar into multiple segments and accumulating points by classification according to these segments. According to Pippenger algorithm, the MSM is rewritten as:

$$\begin{aligned}\sum_{i=0}^{n-1} s_i \cdot p_i &= \sum_{i=0}^{n-1} \sum_{j=0}^{\left\lceil \frac{W_s}{c} \right\rceil - 1} 2^j \cdot s_{i,j} \cdot p_i \\ &= \sum_{j=0}^{\left\lceil \frac{W_s}{c} \right\rceil - 1} 2^j \cdot \sum_{i=0}^{n-1} s_{i,j} \cdot p_i, \end{aligned} \tag{1}$$

where W_s represents the width of the scalars and $s_{i,j}$ represents the j-th partition of scalar s_i with each partition is c-bit. The selected j and c are also referred to as the window and window size, respectively. After classification, the points corresponding to the same scalar segment are distributed to the same bucket identified by the value of that scalar segment. Sequentially, the points in each buckets are accumulated, $\sum_{i=0}^{n-1} s_{i,j} \cdot p_i$ can be rewritten as

$$\sum_{i=0}^{n-1} s_{i,j} \cdot p_i = \sum_{k=0}^{2^c-1} k \cdot B_k = T_j, \tag{2}$$

where the B_k is the accumulated result of points which the corresponding scalar segment $s_{i,j}$ are equal to k. The T_j is the summation of points in all c buckets of j-th segment. This is also the core of bucket method.

Therefore, the bucket method can be divided into three stages:

- **Bucket Classification and Accumulation Phase:** Classify the points into the corresponding buckets according to the scalar segment $s_{i,j}$ and accumulate the points in each bucket to obtain B_k. In this phase, $\frac{W_s}{c}(n - 2^c + 1)$ PADD are required.
- **Bucket Aggregation Phase:** It can be expressed as $\sum_{k=0}^{2^c-1} k \cdot B_k$. It sums points in all buckets with the same j. In this phase, $\frac{W_s}{c}(2^{c+1} - 3)$ PADDs are required.
- **Finale Result Aggregation Phase:** It can be expressed as $\sum_{j=0}^{\left\lceil \frac{W_s}{c} \right\rceil - 1} 2^j \cdot T_j$. In this phase, the points in each segment are doubled and accumulated which requires $(W_s - c + \frac{W_s}{c} - 1)$ PADDs.

According to the discussion above, in three phases, the bucket classification and accumulation phase is directly related to the size of MSM and costs most of the resources and time. The cost of the other two phases is almost negligible taking only 1% of the total [7]. The efficiency of bucket classification and accumulation phase greatly affects the speed of MSM. Therefore, this phase is what we focus on in this paper.

III. Architecture

In this section, we delineate the hardware architecture of ReZK. We first introduce the reconfigurable core that supports

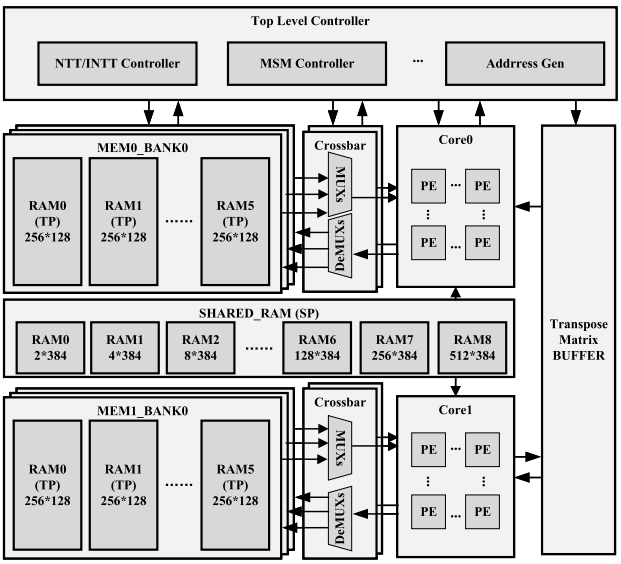


Fig. 6. The top level architecture of ReZK.

different bit-width NTT/INTT or PADD modes. We propose a pipelined reconfigurable modular adder with variable bit-widths supports. Beyond the reconfiguration of the arithmetic unit, the on-chip memories are reused as much as possible to improve resource utilization.

A. Top-Level Architecture

As depicted in Fig. 6, the overall accelerator comprises two reconfigurable cores, two groups of memory arrays (MEM0-MEM1), a shared single-port RAM, a reusable transpose buffer, a crossbar, and independent controllers for NTT and MSM, resepectively. As the basic unit of reconfigurable cores, reconfigurable PEs can be configured as butterfly operation, modular multiplication or modular addition with variable bit-width support. Built upon reconfigurable PEs, the reconfigurable core is tailored to different modes with variable functions, sizes, and bit-widths. For instance, the core exhibits capability in processing 1D-NTT/INTT and pipelined PADD regardless of the chosen bit-width. In NTT mode, the supported size is adjustable, facilitating the realization of 2D-NTT with support for multiple sizes at the upper level of the architecture. In ReZK, we employ two reconfigurable cores, in which each core contains 10 reconfigurable PEs and 3 reconfigurable modular multipliers.

The on-chip memory array also occupies a considerable area, and whether it can be reused has a great impact on resource utilization. In ReZK, we reuse MEM0 and MEM1 as twiddle factor storages or buckets and FIFO, in 2D-NTT and MSM bucket classification stage, respectively. However, the depth of memories needed for NTT/INTT and MSM differs. For NTT/INTT, it depends on the 1D-NTT size, while for multiple-bank buckets in MSM, it is determined by the size of buckets. Therefore, for reusing efficiency, 256-depth RAM as the basic unit is combined to implement the RAM with greater depth. Moreover, due to the 128-bit radix of reconfigurable PEs and data path, the width per RAM is thus set as 128. According to the required memories for NTT/INTT and MSM,

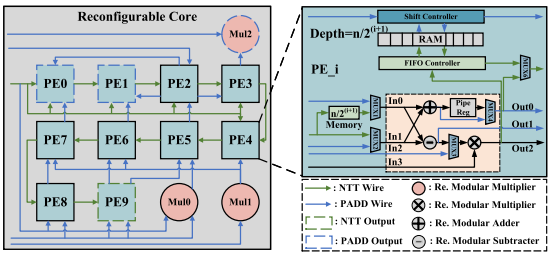


Fig. 7. The architecture and dataflow of reconfigurable core.

MEM0 and MEM1 are split into 10 banks with six 256-depth and 128-width RAMs per bank. Additionally, two synchronous and parallel modules need the same twiddle factor. Therefore, the shared single port RAM is employed simultaneously by two cores to store 1D-NTT twiddle factors. To alleviate the bandwidth pressure in NTT mode, the pipelined reconfigurable transpose matrix buffer with multiple bit-width support is employed. Moreover, this transpose buffer can also be configured as FIFO to be reused in MSM mode.

A distinct disparity exists between the NTT and MSM algorithms, rendering the reuse or reconfiguration of control logic challenging and inefficient. Nevertheless, when compared with computational and memory units, the area occupied by this component is negligible. Consequently, we opt for two controllers, dedicated to NTT and MSM, respectively. In diverse computations, the pertinent controller is activated to govern data scheduling and establish the data path through a configurable crossbar. To minimize power consumption, the clock of the inactive controller is deactivated.

B. Reconfigurable Processing Unit

As shown in Section II, required operations in 1D-NTT/INTT and PADD are significantly distinct. In NTT/INTT, butterfly operation and delay FIFOs are configured as processing unit in each stage. While in pipelined PADD, basic arithmetic units, such as modular multiplication, modular addition, and shifting buffer are needed. Therefore, we propose a unified reconfigurable PE in Fig. 7 to deploy two algorithms' basic unit efficiently.

The arithmetic circuit in PE can be configured as a pipelined butterfly unit or independent Montgomery modular multiplier and modular adder to adapt requirements in NTT/INTT and MSM. It should be noted that Montgomery modular multiplier and modular adder (subtractor) here are all capable of supporting 2-way 256-bit or 1-way 384-bit operation in parallel. These reconfigurable pipelined arithmetic units are the origin by which the ReZK can support multiple bit-widths NTT/INTT or MSM.

1) Pipelined Reconfigurable Montgomery Modular Multiplier Supporting Variable Bit-Widths: For implementing a reconfigurable modular multiplier with supporting variable bit-width, we propose an improved variable bit-width friendly modular multiplication algorithm in [26]. In our improved modular multiplication algorithm, as shown in Algorithm 1, we introduce truncated multiplications, LSB-Mult and MSB-Mult, to reduce the resources. Based on Algorithm 1,

Algorithm 1 Our Improved Modular Multiplication Algorithm Proposed in [26]

Preprocess: $M' = -M^{-1} \bmod R$, $s = \lceil \frac{n}{k} \rceil$
Input: $X, Y \in [0, M)$, $R = 2^n > M$, $\gcd(R, M) = 1$
Output: $Z = XYR^{-1} \bmod M \in [0, M)$

$Z = 0$
$\{Z_0, Z_1, \ldots, Z_{2s-2}\}$ = KO (or KO3) $algorithm(X, Y)$
for $i = 0$ to $s - 1$ **do** ▷ Outer loop
  $Q_{mi} =$ LSB-Mult($Z_0 \bmod 2^k$, M')
  $Z_0 = \lfloor Z_0 / 2^{k-c} \rfloor +$ MSB-Mult(Q_{mi}, M_0)
  $sum_{i/0} = \lfloor Z_0 / 2^c \rfloor + 1$
  for $j = 1$ to $s - 1$ **do** ▷ Inner loop
    $sum_{i/j} = Z_j + Q_{mi} M_j$
  end for
  $Z_0 = sum_{i/0} + sum_{i/1}$
  for $l = 1$ to $2s - 3 - i$ **do**
    if $1 \leqslant l < s - 1$ **then**
      $Z_l = sum_{i/l+1}$
    else if $s - 1 \leqslant l \leqslant 2s - 3 - i$ **then**
      $Z_l = Z_{l+1}$
    end if
  end for
end for
$Z = Z_{s-2} * 2^{k(s-2)} + \ldots + Z_1 * 2^k + Z_0$
if $Z \geq M$ **then**
  $Z = Z - M$
end if
return Z

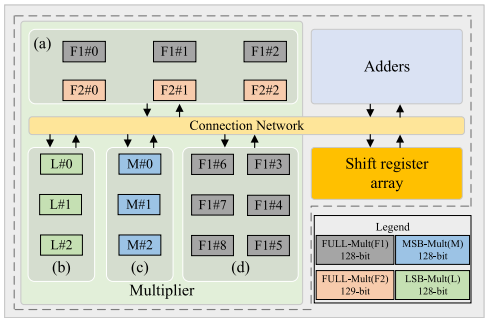


Fig. 8. The reconfigurable modular multiplier with variable bit-widths support [26].

we implement the reconfigurable modular multiplier. Fig. 8 shows the structure of the proposed modular multiplier which consists of multiple full-scale integer multipliers and improved truncated integer multipliers. By reusing integer multiplier instantiated in multiplier array, the reconfigurable modular multiplier, according to configuration, can support 2-way 256-bit or 1-way 384-bit modular multiplication in parallel. In 256-bit, the latency of the pipeline is 14 cycles, while 19 cycles in 384-bit. Please refer to [26] for the details of the reconfigurable Montgomery modular multiplier supporting variable bit-widths.

2) Pipelined Reconfigurable Modular Adder/Subtractor Supporting Variable Bit-Widths: Our reconfigurable modular adder (subtractor) is inspired by [27]. As shown in Algorithm 2, the W-bit inputs of the adder are split into N

Algorithm 2 Low Carry Chain Latency Addition [27]

Input: $P, Q \in [0, 2^W)$, $\mathrm{N} = \lceil log_2(W) \rceil$, $\mathrm{L} = \lceil \frac{W}{N} \rceil$
Output: $Z = P + Q$

for $i = 0$ to $N - 1$ **do**
  $P_i = P[(i+1) * L - 1 : i * L]$
  $Q_i = Q[(i+1) * L - 1 : i * L]$
end for
$\{C_0, Z_0\} = P_0 + Q_0$
for $i = 1$ to $N - 1$ **do**
  $\{C_i^0, Z_i^0\} = P_i + Q_i + 0$
  $\{C_i^1, Z_i^1\} = P_i + Q_i + 1$
  if $(C_{i-1} = 1)$ **then**
    $\{C_i, Z_i\} = \{C_i^1, Z_i^1\}$
  else
    $\{C_i, Z_i\} = \{C_i^0, Z_i^0\}$
  end if
end for
$Z = \{C_{N-1}, Z_{N-1}, Z_{N-2}, \ldots, Z_0\}$
return Z

segments with each of length L-bit ($L = \lceil \frac{W}{N} \rceil$). Then each segment is added. After splitting, the W-bit addition is converted to multiple times L-bit addition, and the carry chain is obviously shortened. However, the design in [27] only supports fixed-bit-width. Based on this method, we propose our reconfigurable pipelined modular adder (subtractor) with variable bit-widths support. Fig. 9 shows an example in which the basic adder is set as 8-bit ($L = 8$). The input values P, Q, and modular M are first split into multiple segments. In 16-bit or 24-bit mode, each segment belongs to the same input while three 8-bit inputs are assigned to three 8-bit segments, respectively. In Stage 1, segments are added similarly to Step 5 to Step 8 of Algorithm 2. Different from Algorithm 2, segment sums and carries, such as C_1, Z_1 and C_2, Z_2 in Stage 2, not only depend on lower segment carry, but also are affected by selected bit-width mode. Stage 3 performs reduction through two's complement of modular. In Stage 4, each segmented result is selected between Z (in Stage 2) and R (in Stage 3) through C and B of the most significant segment which decides the sign bit of the reduction result. This example thus can support 3-way 8-bit, 1-way 16-bit, or 1-way 24-bit modular adder in parallel. Our reconfigurable modular adder is similar to this example apart from 128-bit radix and more segments. The algorithm and structure of the reconfigurable modular subtractor are similar to those of the adder, except that the inputs and each C_i in Stage 1 are inverted. Our fully pipelined reconfigurable modular adder/subtractor can support 2-way 256-bit or 1-way 384-bit computation in parallel with 2-cycle latency.

3) Reusable Memory in PEs: Another critical part of reconfigurable PE is the buffer. In SDC NTT/INTT, FIFO is employed to cache the input and result of the current stage. While in PADD, shift buffers are needed to align the pipeline. These resources are also relatively expensive in PE. To save and reuse these buffers, we employ a two-port SRAM, which can be controlled by the FIFO controller and

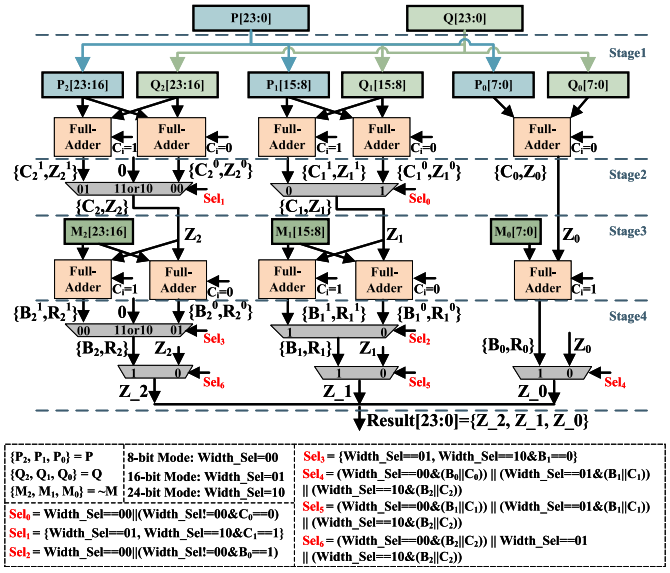


Fig. 9. The example of reconfigurable modular adder with supporting 8, 16, and 24 bits.

Shift Controller, respectively, to store processed data with a 768-bit width. The SRAM depth corresponds to PE's location in the data flow. In other words, this is dependent on the stride of each stage in NTT. For instance, the SRAM depth in PE0 is 512, while it is 256 in PE1. In addition, when controlled by the Shift Controller, the SRAM is reused for shifting data in the pipeline. Furthermore, based on different Montgomery modular multiplier latency under configuration, the delay of the Shift Controller is also adjustable.

Based on reconfigurable PEs, we implement reconfigurable core to support SDC NTT and PADD two algorithms. According to Fig. 7, several PEs can be bypassed, therefore the sizes of NTT, the reconfigurable core can support, vary from 2^6 to 2^{10}. According to 2D-NTT algorithm, twiddle factors used in Step 2 of 2D-NTT need many on-chip memory and precomputation. There is no doubt that this needs remarkable resources and cycles. However, these twiddle factors follow certain rules, that is, most twiddle factors can be updated on-the-fly through computation based on preloaded twiddle factors and twiddle factor seeds. Therefore, the modular multiplier Mul0 is employed to calculate INTT and update the twiddle factors used in Step 2 of 2D-NTT when twiddle factors are read into the core. This means that we don't need extra cycles to read and update them repeatedly.

For PADD in MSM mode, we implement a strongly unified addition algorithm [28] based on Edwards Curves in $\mathbb{G}_1$ group $(x^2 + y^2 = c^2 * (1 + dx^2y^2))$ as shown in Table III, which works whether two points are the same or not. Moreover, the points are represented in projection coordinates $[X : Y : Z]$, which can avoid expensive modular inverse computation.

By analyzing the PADD operation, the overall data flow can be roughly divided into four parallel calculations. Moreover, more data need to be delayed at the end of the pipeline, thus we map the dataflow from the bottom to the top of the reconfigurable core to utilize more memories in top area PEs as shown in Fig. 7.

TABLE III
THE FORMULAS AND MULTIPLICATION AND ADDITION COST OF UNIFIED POINT ADDITION

Cost	Step	Processor 1	Processor 2	Processor 3	Processor 4
3M+1A	1	R_1=$X1$+$Y1$	R_2=$Y1$*$Y2$	R_3=$X1$*$X2$	R_4=$Z1$*$Z2$
1M+3A	2	R_5=$X2$+$Y2$	R_6=R_1*R_2	R_7=R_2+R_3	R_8=R_2-R_3
3M+1A	3	R_9=R_6-R_7	R_{10}=R_4*R_8	R_{11}=R_2*R_3	R_{12}=R_4*R_4
2M+2A	4	R_{13}=R_4*R_9	R_{14}=d*R_{11}	R_{15}=R_{12}-R_{14}	R_{16}=R_{12}+R_{14}
4M	5	R_{17}=R_{15}*R_{16}	$X3$=R_{13}*R_{15}	$Y3$=R_{10}*R_{16}	$Z3$=c*R_{17}

Given a fully pipelined modular multiplier and adder that supports variable bit-widths, the reconfigurable core can handle different bit-widths as long as being constructed according to fixed-bit-width NTT/INTT and PADD data path. This allows the reconfigurable core based on the discussed structure above can process NTT/INTT and PADD in different bit-widths in parallel in a manner similar to multiple data streams.

In summary, the NTT operation utilizes 9 PEs and 1 modular multiplier, while the PADD operation utilizes all resources except a few modular adders that take up very little area. Moreover, based on the characteristics of basic arithmetic units, the reconfigurable core can support 2-way 256-bit and 1-way 384-bit NTT, INTT, or fully pipelined PADD in parallel.

IV. DATAFLOW IN NTT AND MSM MODE

In this section, we elaborate data flow of the ReZK across various modes. Firstly, we employ on-line updating twiddle factors, address generator, and pipelined reconfigurable transpose matrix buffer to improve the efficiency of bandwidth and memory usage. Secondly, we propose modularized MSM to meet variable bit-widths and improve the on-chip memory reusing.

A. ReZK in NTT Mode

In NTT/INTT computation mode, the reconfigurable cores are configured to NTT mode. As discussed in III-B1 and III-B2, the reconfigurable modular multiplier and adder, as the fundamental unit, are all fully pipelined under different bit-widths. Therefore, for processing different bit-widths, no extra work is needed except constructing data paths the same as fixed bit-width NTT. According to modes supported by reconfigurable modular arithmetic units, each core can process 2-way 256-bit, or 1-way 384-bit NTT/INTT in parallel. In each bit-width mode, the data processed in parallel during each cycle are concatenated and fed as input. The ultra amount of data demands per cycle, such as 512 bit/cycle in 256-bit mode, will pose a significant challenge to the bandwidth of off-chip memory.

To alleviate the bandwidth pressure, we employ SDC NTT architecture in reconfigurable cores. However, the bit-width of input data per cycle will be up to 1024 if the ReZK works in 256-bit mode. When working in 800 MHz clock, the bandwidth needed is up to 100 GB/s. The bandwidth pressure will be greater when twiddle factors are loaded together with polynomial coefficients.

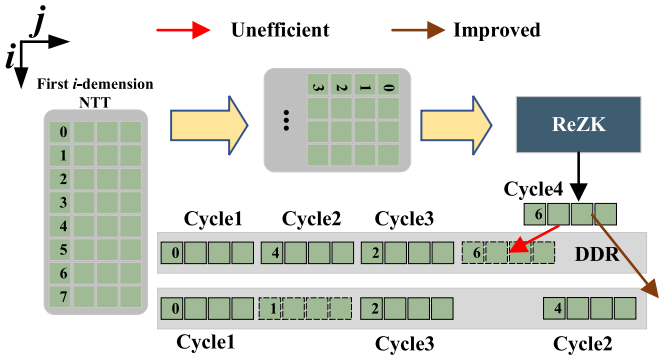


Fig. 10. The example of improved output pattern of ReZK with $I = 8$ and $n = 4$.

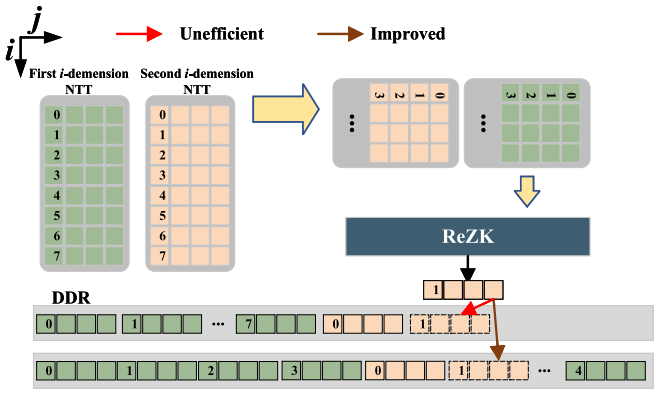


Fig. 11. The example of final version output pattern of ReZK with $I = 8$ and $n = 4$.

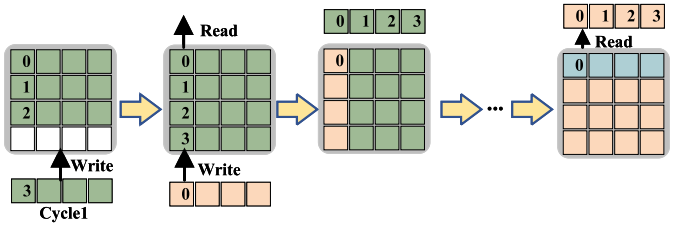


Fig. 12. The example of pipelined reconfigurable transpose matrix buffer with 4×4.

In Step 1 and Step 3 of 2D-NTT, the needed twiddle factors are only related to the configured size and stages in SDC NTT. The twiddle factors between two same dimension NTTs are exactly the same. To address this issue, we employ reusable memories to preload twiddle factors used in Step1. These twiddle factors do not need to be updated until finishing the current dimension. Moreover, these twiddle factors can be shared between two reconfigurable cores.

In Step 2, total $I \times J$ twiddle factors are needed. When I and J are several thousand or more larger, the memories required for these twiddle factors will be quite large. However, as one can see from 2D-NTT flow, the several I-size NTTs are processed sequentially. And the following twiddle factors can be calculated from the previous one. Therefore, the twiddle factors are updated on-the-fly. In our work, the ReZK can process multiple NTT in parallel. Taking 256-bit mode as an example, we only need to preload $4 \times I$ twiddle factors ($\left[\omega^0, \omega^{1j}, \omega^{2j}, \ldots, \omega^{(I-1)j}\right], j \in [0, 3]$) and twiddle factor seed ($\left[\omega^0, \omega^{1*4}, \omega^{2*4}, \ldots, \omega^{(I-1)*4}\right]$). While the first set of 4-way NTT calculations is being performed, the twiddle factor is used and updated by the pointwise product between two vectors simultaneously to $\left[\omega^0, \omega^{1j}, \omega^{2j}, \ldots, \omega^{(I-1)j}\right], j \in [4, 7]$ and written back to MEM0 and MEM1 for use by the next set of 4-way NTT. This also works in 384-bit NTT calculations. For each core, the required twiddle factor memories in this step are related to the bit-width mode, parallelism, and size of NTT/INTT. Clearly, the worst-case scenario occurs when the NTT/INTT bit-width mode and size are set to 256-bit and 1024. In this case, the on-chip memory with a capacity of $(4 + 2) \times 1024 \times 128$-bit is required. Aiming to more efficient scheduling across different bit-width mode, we select 128-bit as the width of each RAM.

After i-dimension NTTs, according to the NTT algorithm, the result is in a reversed order. If these results are directly output to DDR, it will result in an inefficient reading procedure when processing j-dimension NTT due to the row-major data layout in the DDR. Therefore, as shown in Fig. 10, we output an offset address synchronously with the results so that they can be written back to the DDR in normal order. However, for j-dimension NTT, there still is a large stride to read data in the DDR. Thus, as shown in Fig. 11, when the result of i-dimension NTT is written back to the DDR, they are assigned a larger stride address. Each l-cycle output of two reconfigurable cores will be written to a continuous space in normal order. The stride between each l cycle output is J/l. Here l is the parallelism of two reconfigurable cores in NTT mode, for example, l is 4 in 256-bit mode. Moreover, the output of the next i-dimension is also written back in this way and intersects with the I/l spaces of the previous i-dimension NTT output like two "combs". Therefore, two l-cycle outputs, that belong to the same j-dimension NTT, from two continuous i-dimension NTTs are stored in adjacent space. In Step 3 of 2D-NTT, bandwidth usage efficiency is further improved.

In the above method, the output per cycle of l-way NTT is row-major stored. This will result in a l-stride reading for the j-dimension NTT. In [7], a transpose matrix is introduced to resolve a similar problem. However, the transpose matrix in this work does not support variable bit-width and is not fully pipelined which will introduce bubbles and increase cycles. In our work, we introduce a reconfigurable transpose matrix buffer with variable bit-width support to efficiently utilize the bandwidth of off-chip memory. As shown in Fig. 12, we propose a pipelined transpose matrix buffer in [29]. When the transpose matrix buffer is full, new data is written in another direction following the new space that appears due to read data. However, the design in [29] only supports one kind of bit-width. Above that, the transpose matrix buffer is improved in this paper to support transposing in three kinds of bit-width and can be reconfigured as FIFO for paired points in MSM. In this buffer, we employ 64 (8×8) 128-bit registers.

B. ReZK in MSM Mode

Constructed with fully pipelined modular arithmetic units, the reconfigurable cores, in MSM mode, can process point

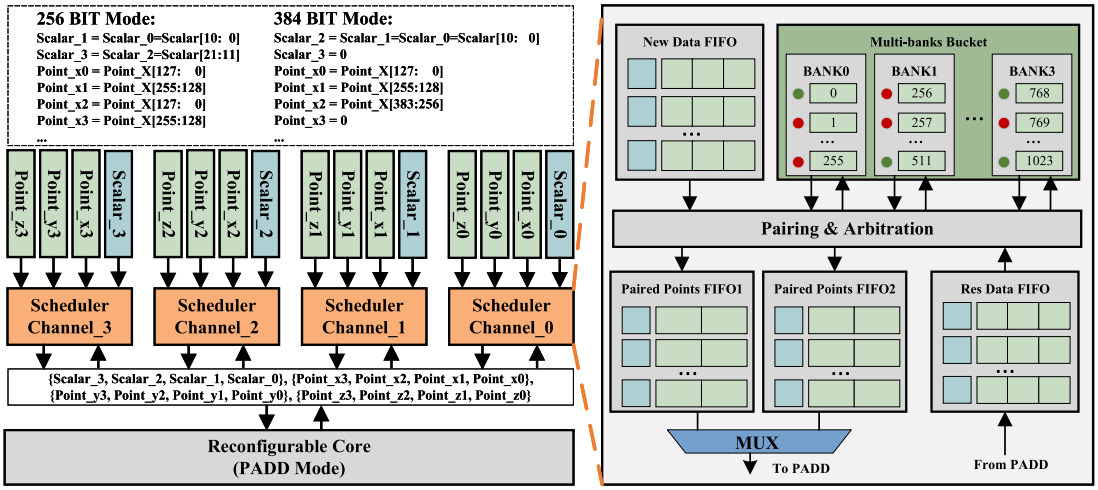


Fig. 13. The modularized MSM with multiple bit-width support.

addition of multiple data streams in parallel and in a pipelined manner. Each core can process 2-way 256-bit or 1-way 384-bit point addition. The bucket classification and accumulation phases of several windows can thus process point addition through the same reconfigurable core in parallel.

Unlike fixed bit-width MSM, the MSM with variable bit-width support can be adapted to more EC after simple configuration. However, the required scheduler in the reconfigurable MSM is more complex. For example, there are different data paths and memory sizes such as FIFOs and buckets for various bit-width. More multiplexers and large-size memories that can not be reused will be introduced if a scheduler designed for fixed bit-width is directly deployed here. The design complexity, critical path, and resource utilization efficiency will all be affected negatively. Therefore, we propose a modularized MSM with multiple bit-widths support in Fig. 13 which can combine multiple channels according to bit-width mode and efficiently reuse expensive on-chip memories in each channel. Each reconfigurable core (in PADD mode) and corresponding scheduler can support 2-way 256-bit or 1-way 384-bit MSM.

As shown in Fig. 13, we implement a 4-channel scheduler for one PADD with each scheduler channel consisting of homogeneous pairing and arbitration logic and same-size buffers. According to the Pippenger algorithm, the pairing and arbitration in each channel are all based on scalars newly loaded from input and scalars backed from the PADD. Moreover, our reconfigurable core in PADD mode is capable of processing points and scalars in parallel and pipelined. It means that scalars backed to two channels in one cycle are the same as long as the scalars sent to the PADD from these two channels in one cycle are the same. Thus any two channels will synchronously pair and arbitrate scalars and synchronously send scalars to and receive scalars from the PADD when the newly loaded scalars are the same. These homogeneous scheduler channels thus can be combined to process larger bit-width MSM when are fed the same scalar.

The kinds of bit-widths that can be supported depend on the bit-width of on-chip memories in each scheduler and the bit-width of PADD. According to the width of reconfigurable cores and reusable SRAM used as buckets, the point bit-width each scheduler channel can process is 128-bit. In 256-bit mode, every two scheduler channels are fed the same scalar. Every dimension of the point coordinate is split into two segments with each segment having a width of 128-bit. The lower segment and higher segment are fed to two channels, respectively. Therefore, four channels can process 2-way 256-bit MSM in parallel. When the lowest three channels are assigned the same scalar, our modularized MSM can process 1-way 384-bit MSM. In our design, there are two reconfigurable cores and 8 scheduler channels in total. In 256-bit mode, the ReZK supports 4-way MSM computation, and 2-way MSM computation in 384-bit mode.

Another point that should be noted is that the points are shared between multiple ways, while the scalars for each way are different. As shown in Fig. 13, the lower way (Channel_0 and Channel_1) and the higher way (Channel_2 and Channel_3) are loaded with different scalars, however, share the same point. Moreover, it still works for ways belonging to different PADDs. By this approach, the efficiency of MSM and bandwidth can be significantly improved.

The shared points between multiple ways result in a new problem. Due to different scalars among each way, the schedulers between different ways are not synchronous. It is possible that some ways are unready to get data from input or PADD. Waiting for each other in multiple ways will greatly reduce efficiency. Thus new data FIFO and result data FIFO are deployed in each scheduler channel to alleviate this situation. To relieve writing or reading conflicts, we set a bucket array with 4 banks. Each bucket has a flag bit to demonstrate the validity of the point. Based on the validity of the flag bit, the pairing and arbitration logic determines whether to read data from the buckets and pair it with points from the new data FIFO and result data FIFO or to write point data from these two FIFOs into the buckets. After successfully paired, the points are first written to Paired Points FIFO1. When two groups of points are paired simultaneously, one group point among them is written to Paired Points FIFO2. This approach will significantly reduce bubbles in the PADD and

improve the pairing efficiency. When conflicts arise between points in the new data FIFO and the result data FIFO, such as accessing the same bucket bank, the points from new data FIFO take priority unless the result data FIFO is about to exceed a threshold that could cause a deadlock. This also minimizes the waiting probability in multiple ways and thus improves the efficiency.

In ReZK, we set the window size to 11-bit. By employing signed scalars [11], the number of point buckets reduces from 2048 to 1024. Thus, the on-chip memory capacity for buckets in each scheduler channel is $1024 \times 3 \times 128$-bit as each point contains 3 coordinates. After splitting into 4 banks, the RAM for coordinate in one dimension is 256-depth and 128-width, requiring 4×3 RAMs per channel. To construct 256-entry result data FIFO based on RAM, we add three more 256-depth and 128-width RAMs. The number of RAMs in each channel comes to 5×3. Therefore, for one reconfigurable core with 4 scheduler channels, there are 20×3 RAMs in total. When classifying these memories into 10 banks, each bank contains six 256-depth and 128-width RAMs. To be reused, these 256-depth RAMs are concatenated to 1024-depth in NTT/INTT mode.

After bucket classification and accumulation in ReZK, points in each channel bucket are uploaded to the host to perform bucket aggregation and result aggregation for which cost is negligible.

V. Results

In this section, we implement and evaluate our design in terms of area and reusability in different modes. Then we evaluate the performance of the ReZK in different configurations and perform comparisons with previous works. Last, we evaluate the end-to-end proof generation performance of real-world applications on our design.

A. Implementation Results

We implement our design in Verilog HDL and synthesize it using Synopsys Design Compiler with a 28nm technology. With the clock constrained as 800 MHz, the area and reusability of the ReZK are shown in Table IV.

The total area of our design is 27.97 mm^2. As one can see, two reconfigurable cores consume the most cost and account for about 21.5 mm^2, 76.9% of the total area. Among them, 78.7% of resources are reusable for both NTT and MSM mode. These resources cost 16.92 mm^2, 60.5% of the overall design. Compared to [20] that also implements 4 pipelined 254-bit PADD PEs costing 20 mm^2, our design achieves better resource-saving. Apart from computation logic, on-chip memories also take up most of the area. The MEM0 and MEM1, used as twiddle factors storage in NTT mode or intermediate points storage in MSM mode, occupy 20.5% area. By taking reusable transpose matrix buffer into account, the total resources that can be reused are about 81.4% of the overall design. Due to completely different schedule methods, the controllers corresponding to NTT and MSM are separately implemented. According to configuration, the corresponding controller is enabled to construct the data path. The area cost by two controllers is smaller than 1%. This greatly increases the flexibility of this work at a lower cost.

TABLE IV
Area Consumption and Reusability of the ReZK

Subsystem		Area(mm^2)	Reusable
Reconfigurable Cores	PEs	16.917(60.5%)	YES
	Others	4.588(16.4%)	NO
Memories	MEM0 and MEM1	5.732(20.5%)	YES
	Shared Memory	0.447(1.6%)	NO
	Transpose Matrix Buffer	0.113(0.4%)	YES
NTT Controller		0.012(0.04%)	NO
MSM Controller		0.145(0.52%)	NO
Others		0.019(0.07%)	NO
Overall		27.97	-

B. Comparisons in Different Configurations

According to the supported configuration, we generate scalars and points data and evaluate our design as follows. In NTT, we primarily test the performance of the 256-bit mode in which the commonly used ECs in ZKP work, such as BN254, BLS12-377, and BLS12-381. Additionally, we exhibit the NTT performance in 384-bit mode. In NTT evaluation, the size of NTT is verified from 2^{14} to 2^{20}. It should be noted that the size supported by this work can be enlarged according to higher dimensions algorithm and corresponding configuration. In MSM mode, we evaluate two bit-width modes, 256-bit for BN254 and 384-bit for BLS12-377 and BLS12-384 curves. The size of MSM is verified from 2^{14} to 2^{26} in our evaluation.

We show various metrics to evaluate this work, such as Time ($\mu s/ms$), Frequency (MHz), Area (mm^2/LUTs&DSPs), and Power (W). For a fair comparison with works on the FPGA platform, we also introduce the number of Clock Cycles (CCs) as a performance metric. As shown in Table V and Table VI, we present the performance of each work in terms of CCs and time, along with the speedup of our design compared to other works. For a more comprehensive and fair comparison with other works based on ASIC platform, we introduce PPAP (Performance×Power×Area) and Energy Efficiency (Operations/Power) metrics to measure all ASIC works. In PPAP metric, the performance means the average consumed time for one coefficient or point computation in NTT/INTT and MSM, while the Operations in Energy Efficiency metric represents the average number of NTT/INTT or MSM operations per second. Therefore, a smaller PPAP value indicates better performance, while higher Energy Efficiency is preferable.

1) NTT Mode: In ReZK under NTT mode, four pipelines are used in 256-bit mode, while 2 pipelines in 384-bit. The performance of our work in NTT mode and comparison with other works are summarized in Table V.

Among works shown in table, only SAM [9] is based on Alveo U250 FPGA platform. In SAM, the arbitrary-sized NTT is decomposed into fixed-sized NTT kernels to flexibly

TABLE V
PERFORMANCE (IN MICROSECONDS) OF THE REZK IN NTT MODE AND COMPARISON WITH OTHER WORKS

Design & Platform	Freq. (MHz)	Area (mm^2)	Power (W)	Bit-Width	Number of Clock Cycles(Speedup in CCs)/Time(Speedup in Time)						PPAP †	Energy Effi. ‡
					2^{14}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}		
SAM [9] AlveoU250	100	594K /6.7K	-	256	-	124K(3.7×)/ 1240(29.6×)	-	-	-	1.3M(2.4×)/ 12610(19.2×)	-	-
GZKP [13] TeslaV100	-	-	-	256	-/50(4.6×)	-/90(2.1×)	-	-/280(1.7×)	-	-/1070(1.6×)	-	-
PipeZK [7] UMC28	300	15.04	1.36	256	23.8K(2.6×) /76(7.0×)	84.3K(2.5×) /281(6.7×)	181.2K(2.7×) /604(7.3×)	446.7K(3.4×) /1489(9.0×)	1.2M(4.6×)/ 4052(12.2×)	3.3M(6.3×)/ 11000(16.7×)	122.9	122.4
SZKP [20] TSMC22	300	18	4.16	254	7.8K(0.9×) /26(2.4×)	26.1K(0.8×) /87(2.1×)	49.8K(0.7×) /166(2.0×)	95.4K(0.7×) /318(1.9×)	189.3K(0.7×) /631(1.9×)	375K(0.7×) /1250(1.9×)	99.0	181.8
Ours TSMC28	800	27.97	8.5	256	8.6K /10.8	33.5K /41.9	66.6K /83.2	132.3K /165.4	263.9K /329.9	526.6K /658.3	151.8	184.3
	800	27.97	8.5	384	16.9K /21.1	66.3K /82.9	132.2K /165.2	263.5K /329.4	526.2K /657.7	1.1M /1313.7	300.7	93.0

† PPAP($ns * mm^2 * W$)=Performance×Power×Area, where performance means the average consumed time for one coefficient or evaluation value in NTT/INTT.
‡ Energy Effi.(MOPS/W)=Operations/Power is an energy efficiency metric, where the OPS represents the average number of NTT/INTT operations per second.

TABLE VI
PERFORMANCE (IN MILLISECONDS) OF THE REZK IN MSM MODE AND COMPARISON WITH OTHER WORKS

Design & Platform	Freq. (MHz)	Area (mm^2)	Power (W)	Bit-Width	Number of Clock Cycles(Speedup in CCs)/Time(Speedup in Time)							PPAP †	Energy Effi.‡
					2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}		
GZKP [13] TeslaV100	-	-	-	256	-/ 4(28.6×)	-/ 6(10.2×)	-/ 15(6.3×)	-/ 45(4.7×)	-/ 170(4.5×)	-/ 720(4.7×)	-/ 2790(4.6×)	-	-
	-	-	-	384	-/ 4(13.8×)	-/ 7(6.0×)	-/ 20(4.3×)	-/ 62(3.3×)	-/ 240(3.2×)	-/ 1100(3.7×)	-/ 4000(3.3×)	-	-
PipeMSM [11] AlveoU55C	125	-	-	377	-	2.2M(2.4×) /17.6(15.0×)	8.6M(2.3×)/ 68.8(14.7×)	34.1M(2.3×) /273(14.6×)	-	-	-	-	-
BSTMSM [10] AlveoU250	300	732K /5.8K	-	384	-	-	3.9M(1.0×) /13(2.8×)	12M(0.8×) /40(2.1×)	43.2M(0.7×) /144(1.9×)	169.2M(0.7×) /564(1.9×)	672.6M(0.7×) /2242(1.9×)	-	-
SZKP [20] TSMC22	300	26	8.56	254	0.3M(2.5×) /0.93(6.6×)	0.7M(1.5×) /2.4(4.1×)	2.5M(1.3×) /8.3(3.5×)	9.6M(1.3×) /31.9(3.4×)	-	-	-	8.7	3.0
PipeZK [7] UMC28	300	35.34	5.05	256	0.3M(2.7×) /1(7.1×)	1.2M(2.5×) /4(6.8×)	4.8M(2.5×) /16(6.7×)	18.3M(2.4×) /61(6.4×)	-	-	-	10.7	3.3
	300	33.72	4.75	384	1.2M(5.2×) /4(13.8×)	3.3M(3.5×) /11(9.4×)	13.8M(3.7×) /46(9.9×)	55.2M(3.7×) /184(9.9×)	-	-	-	30.6	1.1
Ours TSMC28	800	27.97	8.5	256	0.1M /0.14	0.5M /0.59	1.9M /2.38	7.6M /9.48	30.3M /37.92	121.3M /151.68	485.4M /606.72	2.1	13.1
	800	27.97	8.5	384	0.2M /0.29	0.9M /1.17	3.7M /4.67	14.9M /18.67	59.7M /74.68	239.0M /298.72	955.9M /1194.88	4.2	6.6

† PPAP($\mu s * mm^2 * W$)=Performance×Power×Area, where performance means the average consumed time for one point in MSM.
‡ Energy Effi.(MOPS/W)=Operations/Power is an energy efficiency metric, where the Operations represents the average number of MSM operations per second.

support various sizes. Furthermore, the SAM exhibits more excellent performance than prior FPGA-based NTT accelerators. We also compare our work with GZKP [13], which implements 256-bit NTT on the Tesla V100 GPU platform each with 32GB memory. To the best of our knowledge, the NTT based on a single GPU card in GZKP is state-of-the-art among single-GPU-based NTT acceleration works. PipeZK [7] represents the ASIC-based NTT based on UMC 28 nm technology, employing 4 pipelines to accelerate 256-bit NTT. In SZKP [20], a scalable ZKP accelerator framework is implemented with HLS. In this work, 254-bit NTT with 4 PEs is evaluated based on TSMC 22nm technology.

Taking native advantage of the ASIC platform and excellent pipelined design, our work exhibits higher work frequency compared to SAM. Compared to this work, our work achieves 29.6× and 19.2× enhancement. When scaled to the same frequency or compared in terms of the number of clock cycles, our work still has 3.7× and 2.4× improvement. Compared to GZKP, our work yields speedups from 1.6× to 4.6×. Furthermore, the GPU platform apparently has much more memory and need higher costs and power. Under this performance, if the Energy Efficiency metric is similar to ours, the power is must not higher than 3.3W. Apparently, this is almost impossible to achieve. Here efficiency of the ASIC, compared to the GPU, is obviously demonstrated. Compared to PipeZK, we achieve an average speedup of 9.4× compared to PipeZK. In terms of PPAP, PipeZK achieves 19% improvement compared to ours due to the fact that we cost more resources to support multiple bit-widths and functions. Regarding Energy Efficiency, our design achieves over 1.5× improvement. Compared to SZKP, our work yields an average speedup of 2×. In terms of PPAP, SZKP performs better.

However, SZKP is based on TSMC 22nm technology, while our design is based on 28nm. This difference will directly affect the area and power consumption, thereby impacting the PPAP results. Furthermore, our reconfigurable design is intended to support both NTT and MSM. The resources used for computing NTT actually occupy only a portion of the total resources. Therefore, if taking this into account, the actual PPAP of our design will be better. Additionally, our design achieves better energy efficiency. Apart from the discussion above, our work can also support NTT in 384-bit mode. The performance in this mode is shown in Table V.

2) MSM Mode: In MSM mode, our design can process two kinds of bit-width. In 256-bit mode, there 4 threads are processed in parallel, while 2 threads in 384-bit mode. This design can work under 800 MHz for both bit-width modes. Compared with previous works, it shows obvious advantages in speed, energy consumption, and flexibility. The performance of our work and comparison with other works are shown in Table VI.

For MSM mode, comparisons are performed with GZKP [13], PipeMSM [11], BSTMSM [10], and PipeZK [7]. Similar to GPU-based NTT implementation in V-B1, GZKP also separately implements 256-bit and 384-bit MSM using a single Tesla V100 GPU card. To the best of our knowledge, this work achieves the best performance among those based on a single card. In PipeMSM, Xilinx Alveo U55C is adopted and only one PADD is employed to process MSM based on the BLS12-377 curve. Different from PipeMSM, the BSTMSM implements two 384-bit PADDs in the accelerator. Moreover, the BSTMSM leverages a huge amount of memory as buckets to speed up the performance. PipeZK [7] designs two bit-widths of MSM separately with 256-bit and 384-bit based on the UMC 28 nm platform. For 256-bit design, the PipeZK also supports 4 PADDs, while 2 PADDs in 384-bit. SZKP [20] as a scalable accelerator framework generates a MSM with 4PEs, 8-bit windows, and 2048 points per window. In this implementation, SZKP can support 254-bit.

The GZKP exhibits an excellent acceleration effect. However, this benefits from large memory and high computation frequency of GPU in some way. In 256-bit mode, our work has a 28.6× speedup compared to GZKP in size of 2^{14}. With the increase in size, the speed-up effect decreases. In size of 2^{26}, the speedup over GZKP drops to 4.6×. Similarly, in 384-bit mode, our improvement compared to GZKP varies from 13.8× to 3.3× with the size increasing. The large memory of GPU can improve the efficiency of MSM through supplying more buckets, especially for MSMs with large sizes. In addition, it is worth noting that the power of our work is far lower than GPU. With lower energy consumption, our work exhibits a greater power efficiency than GZKP. Limited by the working frequency, the PipeMSM spends a much longer computation time. Compared to this work, our design achieves 15.6× speedup on average. When comparing the number of clock cycles, our design still achieves 2.3× enhancement. The speed of our design is still slightly faster than PipeMSM in which the number of PADD is scaled to two. Compared to BSTMSM, our work achieves a 2× speedup in 384-bit mode. Our work takes up less time which is important in this computation-intensive scenario. Additionally, in our work, flexibility is also retained to a certain degree. In terms of number of clock cycles, our design exhibits worse performance. On the one hand, this is caused by the pauses across the scheduler channel for sharing input data. However, on the other hand, this point is based on that BSTMSM consumes much more True-Dual-Port RAMs, with the depth up to $2^{c-1} \cdot \left\lceil \frac{W_s}{c} \right\rceil$. When set same window size 11-bit, the memory of BSTMSM is larger 8× than ours. It is almost unacceptable for our design to improve a little performance while consuming more memory which may be up to 50 mm^2. Compared to SZKP, our design achieves an average enhancement of 4.4×. In terms of PPAP and energy efficiency, our design improves 4.1× and 4.4×, respectively. The frequency of PipeZK is limited to 300 MHz. The efficiency of them is affected negatively by low frequency. In 256-bit mode, our work achieves 6.8× speedups on average over PipeZK, while 10.7× speedups in 384-bit mode. In terms of PPAP and Energy Efficiency, our design apparently is better. Moreover, our work utilizes fewer resources. Two kinds of bit-width in our work are implemented in one accelerator at the same time. We will show more flexibility after tape out than PipeZK.

The results presented in Table VI demonstrate that our work achieves obvious enhancement in terms of speed, energy consumption, and flexibility.

C. Overall Performance for Different Workloads

We also evaluate the end-to-end proof generation performance of different ZKP systems and real-world application on the proposed accelerator. We compile and run the baselines on a CPU platform with 2.9-GHz Intel Xeon Gold 6226R processor with 754-GB memory.

1) ZKP Proving System Workloads: In this evaluation, we compile and run benchmarks for two ZKP systems. One of the selected baselines is Arkworks-Groth16 [30], which implements the Groth16 protocol in Rust. In ZK development, Arkworks is one of the two main ecosystems and has been applied to many notable projects, such as Aleo and Mina. The other baseline, the Jellyfish [31] library, implements a Plonk protocol in Rust. In Jellyfish, many of the main computation components are also sourced from Arkworks. Furthermore, both libraries support the BN254 and BLS12-381 curves, with bit-width of 256 and 384, respectively. Therefore, we benchmark both ZKP proving systems on BN254 and BLS12-381, and also deploy the corresponding NTT/INTT and MSM computations to ReZK to evaluate its overall speedup.

In both proving benchmarks, we set the size of circuit or constraints that describe the question to be proved to 2^{20} and randomly generate circuits. For generating Plonk proof, our accelerator achieves an average speedup of 4×. When considering only the NTT/INTT and MSM computations, the speedup exceeds 100×. This is because, in the proving phase, the NTT/INTT and MSM computation account for 70-80% of the total time. The other computation, such as polynomial multiplication, is processed on the CPU due to our design has not focused on these parts. Therefore, the computation time of the unaccelerated parts on the CPU dominates proof generation

TABLE VII
ZKP PROVING PERFORMANCE ON REZK (IN SECONDS)

Protocol	ECs	CPU			**Ours**			(I)NTT&MSM	Proof
		(I)NTT	MSM	Proof	(I)NTT	MSM	Proof †	Speedup	Speedup
Plonk	BN254	1.71	16.66	25.9	0.014	0.123	6.95	134.1×	3.7×
	BLS12-381	1.76	28.7	38.9	0.014	0.242	8.70	119.0×	4.5×
Groth16	BN254	0.57	6.41	7.18	0.005	0.038	1.44	132.5×	5.0×
	BLS12-381	0.59	11.04	12.13	0.005	0.075	2.70	117.8×	4.5×

† As the unaccelerated portions are computed on the CPU, we take this part of time into account of proof generation time.

TABLE VIII
ZCASH APPLICATION PERFORMANCE ON REZK (IN SECONDS)

Workload	Vector	CPU			**Ours**			(I)NTT&MSM	Overall
	Size	(I)NTT	MSM	Proof	(I)NTT	MSM	Proof †	Speedup	Speedup
Sapling_spend	131071	0.31	1.6	1.95	0.581ms	0.009	0.37	165.9×	5.3×
Sapling_output	8191	0.02	0.21	0.24	0.035ms	0.6ms	0.05	297.2×	4.6×

† As the unaccelerated portions are computed on the CPU, we take this part of time into account of proof generation time.

time, resulting in a relatively smaller overall speedup. These parts can also be accelerated easily to speed up the overall proving phase. We will address these efforts in the future. In terms of Groth16 protocol, compared to CPU, our design achieves 120× speedup for NTT/INTT and MSM computation. For the entire proof, our design achieves a speedup of over 4.5×.

2) Zcash Application: We also use Zcash workloads [6], a real-world application with employing ZKP to provide enhanced privacy protection, evaluating the performance of the ReZK and its enhancement compared to baselines. The librustzcash [15] which implements a Zcash Rust repository, is selected as a baseline with running on the CPU platform. We generate data by benchmarks in this library as workloads to evaluate our design. Among the protocols in Zcash, we select *Sapling* protocol (composed of *Sapling_spend* and *Sapling_output*) as our workload due to its greater efficiency, security, and flexibility. In these workloads, the proof generation time dominates the transaction flow, making other computations negligible. In this evaluation, the BLS12-381 curve is employed.

Similar to discussed above, the speedup of NTT/INTT and MSM computation is up to 165.9-397.2×. The overall speedup for generating proofs drops to 4.6-5.3×. Similar to the evaluation of ZKP proving systems, the remaining unaccelerated portions dominate the total time after applying our design. We will address these issues moving forward.

VI. CONCLUSION

This paper introduces ReZK, a highly reconfigurable ZKP accelerator capable of concurrently supporting NTT and MSM computations across multiple bit-widths. In NTT mode, ReZK achieves parallel computation of 4-way 256-bit, and 2-way 384-bit NTT or INTT operations, accommodating sizes ranging from 2^{12} to 2^{20}. In MSM mode, ReZK facilitates parallel computation of 4-way 256-bit and 2-way 384-bit MSM operations. Notably, the design emphasizes resource reuse. Most resources are reused in both functions and bit-width. Experimental results demonstrate that ReZK maintains high flexibility without compromising performance, which is particularly beneficial for ASICs.

REFERENCES

[1] D. Wang, J. Zhou, A. Wang, and M. Finestone. (2018). *Loopring: A Decentralized Token Exchange Protocol*. [Online]. Available: https://loopring.org/resources/en_whitepaper.pdf
[2] Matter Labs. (2021). *Zksync Rollup Protocol*. [Online]. Available: https://github.com/matter-labs/zksync/blob/master/docs/protocol.md
[3] P. Labs. (2017). *FileCoin: A Decentralized Storage Network*. [Online]. Available: https://filecoin.io/filecoin.pdf
[4] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 839–858.
[5] C. Fromknecht, D. Velicanu, and S. Yakoubov, "CertCoin: A Namecoin based decentralized authentication system," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep., 2014, vol. 6, pp. 46–56.
[6] E. B. Sasson et al., "ZeroCash: Decentralized anonymous payments from Bitcoin," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 459–474.
[7] Y. Zhang et al., "PipeZK: Accelerating zero-knowledge proof with a pipelined architecture," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 416–428.
[8] Ingonyama. (2021). *ZPU: The Zero-Knowledge Processing Unit*. [Online]. Available: https://www.ingonyama.com/blog/zpu-the-zero-knowledge-processing-unit
[9] C. Wang and M. Gao, "SAM: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Oct. 2023, pp. 1–9.
[10] B. Zhao, W. Huang, T. Li, and Y. Huang, "BSTMSM: A high-performance FPGA-based multi-scalar multiplication hardware accelerator," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2023, pp. 35–43.
[11] C. F. Xavier, "PipeMSM: Hardware acceleration for multi-scalar multiplication," *Cryptol. ePrint Arch.*, Aug. 2022. [Online]. Available: https://eprint.iacr.org/2022/999

[12] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "FPGA acceleration of multi-scalar multiplication: CycloneMSM," *Cryptol. ePrint Arch.*, Oct. 2022. [Online]. Available: https://eprint.iacr.org/2022/1396

[13] W. Ma et al., "GZKP: A GPU accelerated zero-knowledge proof system," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Jan. 2023, pp. 340–353.

[14] T. Lu et al., "CuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs," *Cryptol. ePrint Arch.*, 2022.

[15] Komodo Platform. (2024). *Librustzcash*. [Online]. Available: https://github.com/KomodoPlatform/librustzcash

[16] S. Garg, A. Goel, A. Jain, G.-V. Policharla, and S. Sekar, "zkSaaS: Zero-Knowledge SNARKs as a service," in *Proc. 32nd USENIX Secur. Symp.*, Anaheim, CA, USA, Aug. 2023, pp. 4427–4444. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/garg

[17] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. 35th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, 2016, pp. 305–326.

[18] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptol. ePrint Arch.*, Aug. 2019.

[19] T. Walton-Pocock. (2019). *PLONK Benchmarks I—2.5x Faster Than Groth16 on MIMC*. [Online]. Available: https://medium.com/aztec-protocol/plonk-benchmarks-2-5x-faster-than-groth16-on-mimc-9e1009f96dfe

[20] A. Daftardar, B. Reagen, and S. Garg, "SZKP: A scalable accelerator architecture for zero-knowledge proofs," in *Proc. 33rd Int. Conf. Parallel Archit. Compilation Techn.*, Long Beach, CA, USA, 2024.

[21] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Proc. Int. Conf. Parallel Process.*, Apr. 1996, pp. 766–770.

[22] W.-L. Tsai, S.-G. Chen, and S.-J. Huang, "Reconfigurable radix-$2^k \times 3$ feedforward FFT architectures," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.

[23] A. Cortes, I. Velez, and J. F. Sevillano, "Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation," *IEEE Trans. Signal Process.*, vol. 57, no. 7, pp. 2824–2839, Jul. 2009.

[24] E. Chu and A. George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Boca Raton, FL, USA: CRC Press, 1999.

[25] N. Pippenger, "On the evaluation of powers and related problems," in *Proc. 17th Annu. Symp. Found. Comput. Sci.*, Oct. 1976, pp. 258–263.

[26] H. Zhou, C. Liu, L. Yang, L. Shang, and F. Yang, "A fully pipelined reconfigurable Montgomery modular multiplier supporting variable bit-widths," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Jun. 6, 2024, doi: 10.1109/TCAD.2024.3410847.

[27] O. Mazonka, E. Chielle, D. Soni, and M. Maniatakos, "Fast and compact interleaved modular multiplication based on carry save addition," in *Proc. 41st IEEE/ACM Int. Conf. Comput.-Aided Design*, Apr. 2022, pp. 1–9.

[28] D. J. Bernstein and T. Lange, "Faster addition and doubling on elliptic curves," in *Proc. 13th Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, Kuching, Malaysia. Cham, Switzerland: Springer, Dec. 2007, pp. 29–50.

[29] C. Liu, D. Tang, J. Song, H. Zhou, S. Yan, and F. Yang, "HMNTT: A highly efficient MDC-NTT architecture for privacy-preserving applications," in *Proc. Great Lakes Symp. VLSI*. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 7–12, doi: 10.1145/3649476.3658734.

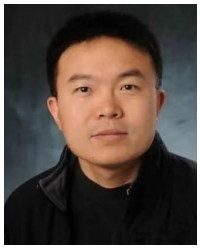
[30] Arkworks. (2024). *Groth16*. [Online]. Available: https://github.com/arkworks-rs/groth16

[31] Espresso. (2024). *Jellyfish*. [Online]. Available: https://github.com/EspressoSystems/jellyfish

Hao Zhou received the B.S. degree in electronic science and technology from Jilin University, Jilin, China, in 2017, and the M.S. degree in electronics and communication engineering from the University of Chinese Academy of Sciences, Beijing, China, in 2020. He is currently pursuing the Ph.D. degree in electronic and information engineering with the School of Microelectronics, Fudan University, Shanghai, China. His research interests include zero-knowledge proof, fully homomorphism encryption, and VLSI implementation of digital systems.

Changxu Liu received the B.S. degree in microelectronics science and engineering from Wuhan University, Wuhan, China, in 2022. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China. His current research interests include hardware-software co-design for privacy-preserving computing applications and digital IC design.

Lan Yang received the B.S. degree in microelectronic science and engineering from Fudan University, Shanghai, China, in 2023, where she is currently pursuing the Ph.D. degree with the State Key Laboratory of Integrated Chips and Systems, School of Microelectronics. Her research interests include homomorphic encryption in privacy-preserving technologies and hardware acceleration.



Li Shang (Member, IEEE) received the Ph.D. degree from Princeton University, Princeton, NJ, USA. He was the Deputy Director and the Chief Architect of Intel Labs, China, and an Associate Professor with CUBoulder, Boulder, CO, USA. He is currently a Professor with the School of Computer Science, Fudan University, Shanghai, China. His research interests include computer systems, human-centered computing, machine learning, VLSI, and EDA, with over 100 publications, multiple best paper awards, and nominations. He was a recipient of the NSF Career Award.

Fan Yang (Member, IEEE) received the B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 2003, and the Ph.D. degree from Fudan University, Shanghai, China, in 2008. He is currently a Full Professor with the Microelectronics Department, Fudan University. His research interests include model order reduction, circuit simulation, high-level synthesis, acceleration of artificial neural networks, acceleration of privacy-preserving computing, and yield analysis and design for manufacturability.