Hardware Acceleration of Number Theoretic Transform for zk-SNARK

Haixu Zhao[1], Dong Ding[2], Feng Wang[1], Pengcheng Hua[1], Ning Wang[1,3], Qin Wu[1,3], Zhilei Chai[1,3*]
[1]*School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi 214122, China*
[2]*School of Internet of Things Engineering, Jiangnan University, Wuxi 214122, China*
[3]*Jiangsu Provincial Engineering Laboratory of Pattern Recognition and Computational Intelligence, Wuxi 214122, China*
Email: {6201910039,6211924040,6213113107,6213113061}@stu.jiangnan.edu.cn,
{8202206032,qinwu,zlchai}@jiangnan.edu.cn

***Abstract*—Zk-SNARK unleashes the great potential of ZKP (zero-knowledge proof) in the blockchain, distributed storage, etc. However, the proof-generation of zk-SNARK is excessively time intensive, making it a challenge to deploy a high-performance zk-SNARK in most real applications. As a result, NTT (Number Theoretic Transform), one of the most time-consuming parts in proof-generation, needs to be accelerated significantly. To address this issue, we propose a novel and efficient "data reordering" technique to enable a highly pipelined architecture, on which an FPGA-based hardware accelerator is designed to support the large-bitwidth and large-scale NTT tasks in zk-SNARK. Our architecture achieves a two-level pipeline: 1) the top-level pipeline is achieved among smaller NTT sub-tasks, which are decomposed from a large-scale NTT task; 2) the bottom-level pipeline is achieved in each sub-task, among butterfly operations with different step sizes. This architecture can effectively reduce the data dependency and memory access requirements, meanwhile, can be flexibly scaled to different scales of FPGAs. To balance computing efficiency and flexibility, the OpenCL equipped with HLS is used to implement the heterogeneous acceleration system. We prototype the accelerator on the AMD-Xilinx Alveo U50 card (UltraScale+ XCU50 FPGA). The evaluation results show that 1) our accelerator shows high scalability for different scales of FPGAs with a stable performance improvement; 2) it performs 1.95× faster than the one in PipeZK; 3) and it achieves 27.98×, 1.74× speedup and 6.9×, 6× energy efficiency improvement than AMD Ryzen 9 5900X single core and 12 cores respectively when integrated into the well-known ZKP open-source project, Bellman.**

***Index Terms*—zk-SNARK (zero-knowledge Succinct Non-interactive Arguments of Knowledge), NTT (Number Theoretic Transform), Hardware Accelerator, FPGA (Field Programmable Gate Arrays), OpenCL**

I. Introduction

Zero-Knowledge Proof (ZKP) [1] refers to the cryptographic protocols by which a prover can convince a verifier that an assertion is true without providing any related information. It is used in many fields, such as digital signature [2], blockchain [3], [4], and distributed storage [5]. Zk-SNARK (zero-knowledge Succinct Non-interactive Arguments of Knowledge) [6]–[8], a class of ZKP protocols, is used to generate short, fixed-length proofs for complex programs and provide fast verification, improving the usability of ZKP in real-world applications. Thus, zk-SNARK has drawn increasing attention in many open-source projects and studies.

Although the proof-verification in zk-SNARK is fast and can be done within dozens of milliseconds, the proof-generation is very complex and time-consuming. To make matters worse, the proof-generation is used frequently and becomes the performance bottleneck of zk-SNARK. Take one of the most popular zk-SNARK instances, Groth16 [9] as an example. Before generating a proof, the Groth16 converts the program to be proved into R1CS and QAP representations [10], which results in several polynomials with millions of coefficients. During the process of proof-generation, these polynomials need to be multiplied and homomorphic encrypted with a large number of operations. In a digital transaction application, it often takes several minutes to process the proof-generation in each transaction [4]. It takes even longer in some other applications, such as distributed storage [5] and outsourced computing [11]. In those scenarios, the proof-generations can take up to 20 to 30 minutes [12].

Through further analysis, large-scale polynomial multiplication is one of the most time-consuming parts in proof-generation [13]. Even if many advanced ZKP protocols [14]–[18] have been proposed, large-scale polynomial multiplication is still inevitable. NTT (Number Theoretic Transformation) and iNTT (inverse Number Theoretic Transformation) are used to accelerate polynomial multiplication. Many applications, such as fully homomorphic encryption [19]–[21], post-quantum cryptography [22]–[24], tend to use NTT and iNTT (inverse Number Theoretic Transformation) to reduce the time complexity of polynomial multiplication. Therefore, many domain-specific NTT hardware and software-hardware collaborative acceleration schemes have emerged [25], [26] in recent years. HEAX [26], an NTT architecture designed for fully homomorphic encryption, achieves high parallelism of NTT but only supports small-bitwidth tasks (up to 54 bits). Mert et al. [25] proposed a polynomial multiplier architecture, which balances time performance between the NTT and I/O operations and obtains a throughput of 800KB/s. However, the maximum input size is limited to 1024. None of the above architectures can support NTT/iNTT tasks with millions of large-bitwidth coefficients, which are required in the proof-generation for zk-SNARK. If we scale up the input size and bitwidth of the below methods as required in zk-SNARK, the area and energy overheads of those accelerators will increase

significantly. Furthermore, the required computation resources for the butterfly operation itself in the NTT module also scale in a super-linear fashion. Both make it inefficient to support NTT in zk-SNARK.

More recently, PipeZK [13] has been proposed to provide high-performance ZKP systems with lower power consumption and response latency. However, the low adaptability of the ASIC-based ZKP or FPGA-based ZKP implemented in VHDL/Verilog makes it less competitive in the era where the algorithms upgrade quickly and frequently. Furthermore, the poor compatibility of the interfaces makes most of the hardware accelerators difficult to be integrated into a real application or open-source project. Therefore, some GPU-based zk-SNARK systems were proposed [27]. Those systems achieve better performance but at the cost of higher power consumption. In addition, the response latencies for executing the NTT on CPUs or GPUs are unstable, which degrades the quality of service of ZKP systems. In response to the above problems, we want to improve the speed and energy efficiency of proof-generation for zk-SNARK as much as possible to maintain good scalability and algorithm adaptability. Therefore, we choose the AMD-Xilinx data center accelerator card and CPU to accelerate the proof-generation heterogeneously.

In this paper, we propose an FPGA-based hardware accelerator of NTT. The OpenCL equipped with HLS (High-Level Synthesis) is used for implementation to provide high flexibility and high compatibility of the system. The main contributions of this paper are summarized as follows:

- We adopt an improved two-dimensional partition method to get the independent smaller sub-tasks of NTT. It improves the parallelism of NTT calculation and reduces the difficulty of designing an NTT accelerator. The accelerator can be scaled for different scales of FPGAs flexibly.
- We design a novel and high-performance hardware architecture for NTT used in zk-SNARK. A "data-reordering" technique is proposed for the butterfly operations in NTT. Thus, a two-level pipeline among sub-tasks and among butterfly operations with different step sizes is implemented. Large-bitwidth modular arithmetic units are also designed especially for Montgomery modular multiplication. Compared with PipeZK, our work can achieve a 1.95× speedup on the same FPGA platform.
- Based on the OpenCL, the FPGA-based NTT accelerator has been integrated into the well-known ZKP open-source project, Bellman. So that, we can further evaluate and improve the performance not only in an isolated module but also in a real system. The accelerator achieves a throughput of 32MB/s, 1.74× speedup, and 6× energy efficiency improvement than a 3.7 GHz AMD Ryzen CPU.

II. Background

A. ZKP and zk-SNARK

ZKP establishes an agreement between the prover and the verifier. The prover can convince the verifier by providing the proof associated with the proposition, without revealing any proposition-related information. Due to three important properties of correctness, zero-knowledge, and succinctness, zk-SNARK gradually replaces the way that the prover and the verifier generate and verify the proof interactively. Zk-SNARK enables the prover to generate a unique proof once according to the polynomial, which in turn can be verified by any verifier.

The object of proof in zk-SNARK is a polynomial equation. It is modeled from the proposition to be proved before generating the proof. First, a polynomial equation of n variables is defined as follows:

$$f(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{n} \sum_{j=0}^{d-1} a_{ij} \cdot x_i^j = 0 \tag{1}$$

where a_{ij} represents the polynomial coefficient, and d means the order of a polynomial. Thus, the program can be described as follows: given a polynomial equation $f(x_1, x_2, \cdots, x_n) = 0$, both the prover and the verifier know the coefficients **a**, but only the prover knows a set of solutions, **x**, and needs to provide a proof to the verifier. The prover needs to prove to the verifier that the prover does hold **x** and meanwhile contains zero information related to **x**. Usually, the polynomial transformation process of the program is performed by an upper-layer application, which is independent of the main process in zk-SNARK. Groth16 [9], as one of the most popular algorithms of zk-SNARK, divides the protocol into three stages shown in Fig. 1:

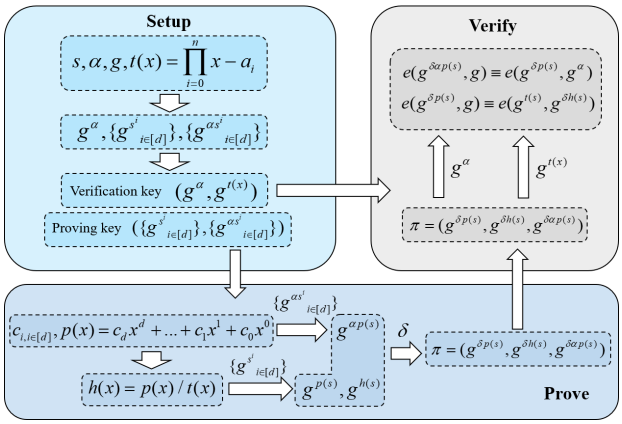


Fig. 1. Groth16 algorithm flow.

- **Setup:** generate proving key and verification key.
- **Prove:** generate proof π.
- **Verify:** verify the proof π.

Usually, the computations are obvious and simple in the **setup** stage and the **verify** stage, however, become much more intensive in the **prove** stage. Among the computations in the **prove** stage, it costs the most to compute $h(x)$ based on the existing polynomials $t(x)$ and $p(x)$. The result of the homomorphic encryption of $\delta p(s)$, $\delta h(s)$, and $\delta \alpha p(s)$ constitute the final proof π.

It is the large bit-width (up to several hundred) and millions of coefficients involved in polynomial multiplication

contribute most to the time-consuming computations in the **prove** stage. More specifically, NTT/iNTT is the major computation of polynomial multiplication. Taking Bellman [28], a popular zk-SNARK library developed in Rust, as an example, the NTT/iNTT in the polynomial multiplication takes a few minutes in total. The detailed computation time of NTT/iNTT at different scales in Bellman is shown in Fig. 2. Therefore, it is necessary to design an efficient NTT computing architecture for polynomial multiplication in zk-SNARK.

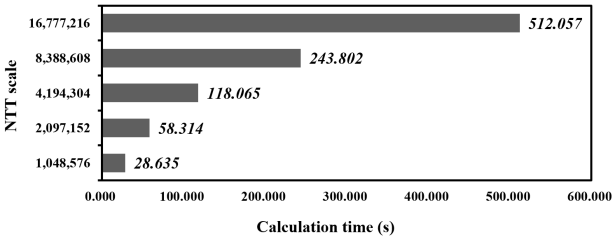


Fig. 2. The calculation time of NTT/iNTT at different scales in Bellman.

B. Number Theoretic Transform

It is common to convert a coefficient representation of a polynomial to a point-value representation for polynomial multiplication. In this way, we reduce the time complexity from $O(n^2)$ to $O(n \cdot \log n)$. The calculation process of polynomial multiplication and its time complexity are shown in Fig. 3. The polynomial multiplication of zk-SNARK is performed on the finite field with q elements, where q is a large prime. As a result, NTT and iNTT are the best choices for interpolation and evaluation.

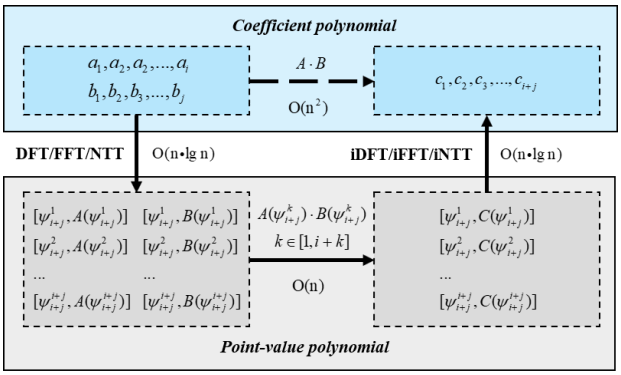


Fig. 3. The relationship and time complexity of *evaluation*, *interpolation*, and *polynomial multiplication* in *point-value representation, and coefficient representation.*

The input of NTT $\mathbf{a}$ is a one-dimensional vector consisting of polynomial coefficients, and the output $\hat{\mathbf{a}}$ is a vector of the same scale, representing the result of the evaluation. The calculation of NTT is defined as follows:

$$\hat{a}_j({\psi_N}^j) = \sum_{i=0}^{N-1} a_i \cdot {\psi_N}^{ij} \tag{2}$$

Where $\mathbf{a}$ and $\hat{\mathbf{a}}$ are vectors of length N; ψ_N is the root of unity over the finite field defined by the prime p; $\hat{a}_j$, a_i and the twiddle factor ${\psi_N}^{ij}$ are all defined over a finite field of bit-width λ. The only difference between the calculation of NTT and iNTT is that the root is the modular inverse of p in iNTT.

Usually, the divide-and-conquer is used to reduce the time complexity of NTT calculation. First, we expand the input vector $\mathbf{a}$ to a length of the nearest power of 2 by padding with zeros. Next, we divide the NTT task into two sub-tasks based on odd and even subscripts of $\hat{a}_j$:

$$\hat{a}_j^+({\psi_N}^j) = a_0 + a_2 \cdot {\psi_N}^j + \cdots + a_{N-2} \cdot {\psi_N}^{(\frac{N}{2}-1)j} \tag{3}$$

$$\hat{a}_j^-({\psi_N}^j) = a_1 + a_3 \cdot {\psi_N}^j + \cdots + a_{N-1} \cdot {\psi_N}^{(\frac{N}{2}-1)j} \tag{4}$$

$$\hat{a}_j({\psi_N}^j) = \hat{a}_j^+({\psi_N}^{2j}) + {\psi_N}^j \cdot \hat{a}_j^-({\psi_N}^{2j}) \tag{5}$$

In this way, the $\psi_{\frac{N}{2}}$ is used to generate the $\frac{N}{2}$ twiddle factors, which are further used in calculating $\hat{a}_j^+$ and $\hat{a}_j^-$. Finally, $\hat{a}_j$ is the sum-up of $\hat{a}_j^+$ and $\hat{a}_j^-$. According to (5), the time complexity of reducing the sub-tasks is $O(n)$. And the time complexity of the recursive implementation of NTT based on divide-and-conquer is as follows:

$$T(n) = 2T(n/2) + O(n) = O(n \cdot \log n) \tag{6}$$

C. Hardware Acceleration Opportunities

When we divide the NTT task in half continuously, butterfly operations with different step sizes will be generated. Butterfly operations are the operations of strided reading and updating 2 elements in NTT. Usually, multiple rounds of butterfly operations with different step sizes are difficult to parallelize, due to their recursive hierarchy, and strict data dependency. However, it is possible to parallelize the butterfly operations with the same step size, as no data dependency exists among them. If we assume the latency of a single butterfly operation is t, for an n-size NTT task, with adequate available threads, the entire NTT task can be completed in $\log n \cdot t$.

However, a GPU cannot create an infinite number of threads, and resources on an FPGA are limited. At present, the hardware accelerators used for NTT can only process a large amount of data or a large data bit-width [21], [22], [29], as shown in Table I. Even PipeZK [13] cannot fully achieve parallelization for NTT tasks of zk-SNARK. In addition, the development methods and hardware platforms of these work hinder the updating and reusing in zk-SNARK application. Naive expansion of these NTT accelerators will result in a nonlinear expansion of the hardware resource usage, which cannot be verified and evaluated. Therefore, balancing time complexity and resource usage in hardware design is the key to NTT acceleration.

III. Proposed Architecture of NTT

A. Large-Bitwidth Modular Arithmetic

Arithmetic operations used in the NTT of zk-SNARK are modular arithmetic defined on large finite fields. Traditionally,

TABLE I
NTT ACCELERATORS OF DIFFERENT SIZES IN RELATED WORK

Work	Size	Bit-width
Chen et al. [22]	1024	31
Öztürk et al. [21]	32768	32
Mert et al. [25]	1024	32
HEAX [26]	16384	54
PipeZK [13]	1048576	768

modular arithmetic consists of basic arithmetic operation and modular operation. The modular operation returns the unsigned remainder of a division. Unfortunately, the hardware implementation of the divisor is expensive, and the cost increases significantly if the inputs have a large bit-width, such as zk-SNARK. Therefore, it is necessary to design efficient hardware units for modular Arithmetic.

For two unsigned integers of n-bit a, b, the result of addition c is no wider than $n+1$ bits, and the result of subtraction d is no wider than n bits. In other words, the range of a and b is $[0, 2^{n+1})$. Since the range of the n-bit module p is $[2^n, 2^{n+1})$, the value of c and d must be less than twice of p. That is, the result of $c \ mod \ p$ and $d \ mod \ p$ are either c and d, or $c-p$ and $d-p$. Therefore, instead of using divisors, we use a combination of comparators, adders, and subtractors to achieve a more resource-efficient and less compute-intensive modular unit for modular addition and modular subtraction.

However, since the output of a basic multiplication has twice bit-width of the modulus, the modular operation in a modular multiplication can only be achieved by a divisor. Therefore, it is necessary to find an alternative way to implement modular operation for modular multiplication efficiently. To resolve it, we replace the traditional modular multiplication with hardware-friendly Montgomery modular multiplication [30] and then encapsulate it into a hardware unit. The calculation steps are shown in Algorithm 1.

Algorithm 1 Montgomery Modular Multiplication

Input: $P, A, B, R = 2^n, P' = -P^{-1} \bmod R$
Output: $C = A \cdot B \cdot R^{-1} \bmod P$
1: $T \leftarrow A \cdot B$
2: $M \leftarrow T \cdot P' \bmod R$
3: $C \leftarrow (M \cdot P + T)/R$
4: **if** $C \geq P$ **then**
5: $\quad C \leftarrow C - P$
6: **end if**
7: **return** C

Montgomery modular multiplication converts the multiplier into a modular-friendly form through the Montgomery reduction, and then achieves modular multiplication through three multiplication operations and several shift operations. Compared with the traditional method, Montgomery modular multiplication increases the number of basic multiplications, however, the most costly division operations are replaced by Montgomery reductions and shift operations, leading to an overall computation latency reduction. Therefore, Montgomery modular multiplication has a significant advantage in the algorithm, but it also faces the problem of how to implement it faster and more resource-saving.

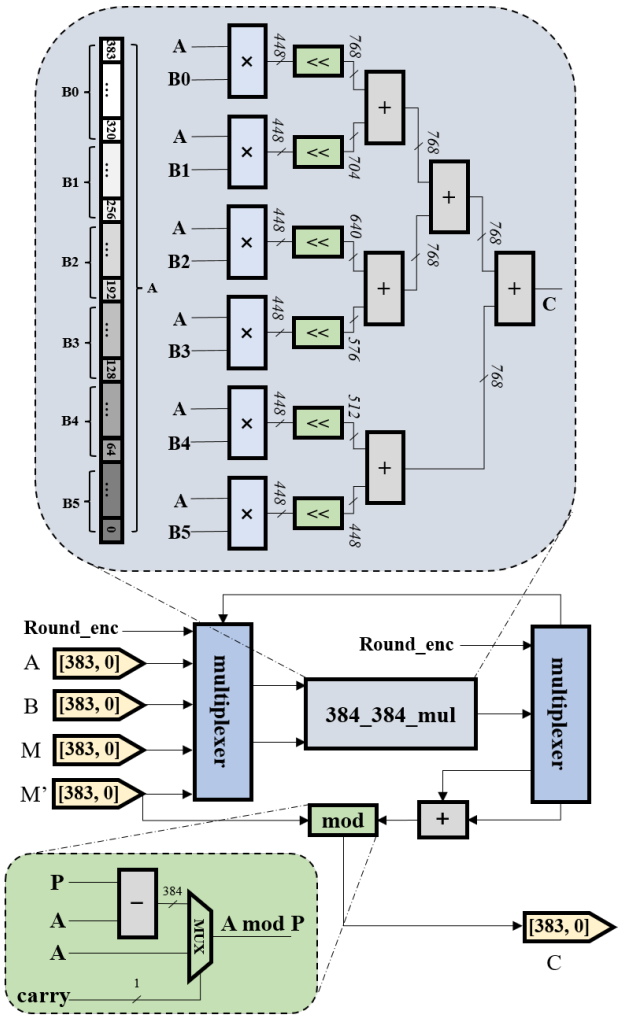


Fig. 4. 384-bit Montgomery modular multiplication unit.

We implement two types of Montgomery modular multiplication units, which can be adapted to the BN128 (256-bit) and BLS381 (384-bit) curves respectively. The 384-bit Montgomery modular multiplication unit is shown in Fig. 4. In contrast, the 256-bit Montgomery modular multiplication unit contains fewer sub-multipliers with lower bit-width in a 256-bit multiplier. The Montgomery modular multiplication unit is composed of a large-bitwidth multiplier and a modulo operator including an adder, a subtractor, and a comparator. Inspired by the Karatsuba algorithm [31]–[33], we do not use plenty of DSP resources to implement a 384-bit multiplier directly, but combine six 64×384 multipliers with shift and addition operations to build a low-latency 384-bit multiplier. This method avoids the problem of cross-SLR routing caused by a single unit utilizing a large number of DSPs. On the one hand, due to data dependency, the three multiplications in the

Montgomery modular multiplication cannot be parallelized. On the other hand, instantiating three 384-bit multipliers uses significant DSP resources. Therefore, we only instantiate one 384-bit multiplier and reuse it through the **allocation** directive in HLS. Although extra two clock cycles are introduced, this approach saves nearly 70% usage of DSP resources. We also apply the same design to the 256-bit Montgomery modular multiplication unit.

To further reduce the computation overhead, we convert the data to its respective Montgomery form on the host side CPU, and use it in further calculations in zk-SNARK, including NTT. In this way, the correctness of the final proof is not affected, but we save considerable Montgomery reduction operations which occur before and after the Montgomery modular multiplication.

B. Computation Partition

We can reduce the time complexity of an NTT task by decomposing a larger-size NTT task into its half size recursively. However, it is impossible to obtain independent sub-tasks of any scale through partition only once. Especially when the NTT input scale is too large, we need to plan the sub-task partition dynamically and control the calculation order strictly, which is difficult for the hardware implementation of the data flow control system. Inspired by two-dimensional partition method [34]–[36], we adopted an improved sub-tasks partition method. In case of huge input size, an NTT task can be decomposed once into multiple sub-tasks. Then these smaller independent sub-tasks can be calculated quickly. In this way, the time complexity of processing an N-size NTT task is $O(n \cdot \log n)$. The partition method is shown in Fig. 5. Each sub-task can be further calculated recursively or iteratively.

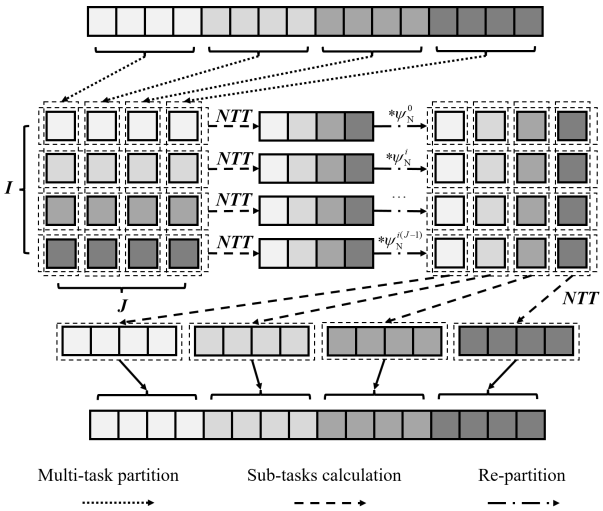


Fig. 5. Computation partition for a large-scale NTT task.

- **Multi-task partition.** Divide an N-size NTT task into J I-size NTT sub-tasks.
- **Sub-tasks calculation.** Use the divide-and-conquer method mentioned in II-B to handle these NTT sub-tasks separately. Multiply the results of the J sub-tasks by twiddle factors $\psi_{\mathrm{N}}^{I(j-1)}$ in turn.
- **Re-partition.** Combine the results of J sub-tasks into a J-by-I array, then re-partition the task into I new sub-tasks of size J.
- **New sub-tasks calculation.** Compute the new NTT sub-tasks recursively and concatenate the outputs into a continuous sequence.

It is difficult to implement an on-chip core to handle a million-size NTT task in zk-SNARK. Therefore, we use the prior partition method to decompose a large-size NTT task into plenty of smaller sub-tasks that can be processed on an on-chip core. Specifically, we decompose an N-size NTT task into $\sqrt{N}$ $\sqrt{N}$-size sub-tasks and design an efficient and reusable $\sqrt{N}$-size NTT core. In most cases, the on-chip resources are adequate to support a $\sqrt{N}$-size (thousand-size) NTT core.

Though the recursive NTT algorithm can successfully reduce the time complexity of an N-size NTT task from $O(n^2)$ to $O(n \cdot \log n)$, the recursive calculations in the algorithm cannot be parallelized. As a further improvement, we convert the recursive calculations into the parallelizable iterative calculations shown in Fig. 6. For example, in an 8-size NTT task, the input vector $\mathbf{a}$ is updated to $\mathbf{a}'$ by a round of butterfly operations with a step size of 4. Then we reduce the step size to its half size and update $\mathbf{a}'$ to $\mathbf{a}''$ by another round of butterfly operations with a step size of 2. We repeat the same process until the step size becomes 1 and generate the final output vector $\hat{\mathbf{a}}$. The butterfly operations in each round are independent and can run in parallel, which well resolved the issue that recursion cannot process in parallel. Therefore, instead of calculating the NTT sub-tasks recursively, we iteratively update the input vector for the next round of butterfly operations with different step sizes.

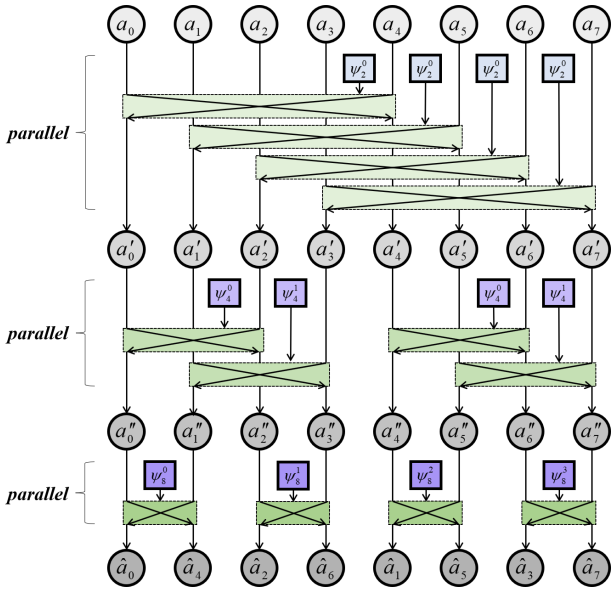


Fig. 6. Iterative computation and butterfly operation in an 8-size NTT.

C. NTT Core and Data-reordering

We design a 4096-size NTT core, which is large enough to handle the sub-tasks divided from the million-size NTT task. Further, the NTT core divides the 4096-size NTT task into 2^n 2^m-size NTT sub-tasks, where $2^n \cdot 2^m = 4096$. As a result, we are able to efficiently implement the pipeline among m rounds of butterfly operations with a step size of 2^i , and further implement the pipeline among 2^n sub-tasks.

The data dependency of the two-level pipeline in the NTT core is shown in Fig. 8. The calculations in the following round can start processing before the completion of all calculations in the prior round. For example, we can perform butterfly operations in the second round (step size=8) once the $\frac{1}{2}$ of tasks in the first round (step size=16) has been completed, and perform butterfly operations in the third round (step size=4) once the $\frac{1}{4}$ of tasks in the second round has been completed, and so forth. Compared to the previous round, the initiation interval of the current round can be reduced to half. We assume that the computation latency of butterfly operations in each round is T, and each butterfly operation unit with a different step size has been instantiated , a 2^m-size NTT sub-task which originally costs mT, can be done in around $2T$, using the designed pipeline among rounds. To Implement asynchronous computation among multiple rounds of butterfly operations, we propose a novel technique named "data-reordering" in Fig. 7.

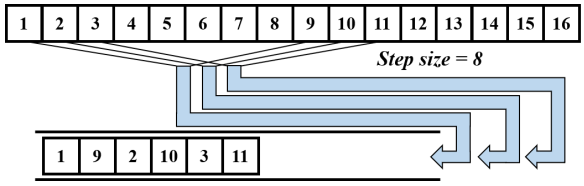


Fig. 7. Data-reordering with a step size of 8.

The iterative NTT scheme requires strided memory accesses in each butterfly operation, and the butterfly operations cross rounds have to run in sequence. The "data-reordering" saves the time spent on the strided memory accesses in a butterfly operation, and, to some degree, helps to achieve the parallelism of butterfly operations in continuous rounds. For example, Fig. 7 shows "data-reordering" in a round of butterfly operations with a step size of 8. We store the outputs of butterfly operations in the current round into a BRAM. At the same time, once adequate data is received in the BRAM, we pick two elements in a fixed step size at a time and send them into a FIFO in order. The process is repeated until all data has been processed. Each round of butterfly operations is no longer executed circularly but is triggered to perform. As long as there are elements in the FIFO, the butterfly operation units connected with it will be triggered to take the elements, calculate, and store the results in local memory according to a specific step. When the data is sufficient to trigger the next round of butterfly operations, it will be filled to the next FIFO. Strictly speaking, "data-reordering" does not save the time spent on strided memory accesses, but utilizes the computation time of the remaining butterfly operations in the prior round to cover it. Thus, a single butterfly operation and the reordering of two output elements can process in parallel. Compared with partitioning BRAM for each round of butterfly operations to achieve multi-port accesses, our proposed method takes fewer resources to implement.

Taking advantage of the "data-reordering", we are able to implement a pipeline among m rounds of butterfly operations with different step sizes. Based upon it, we design a tightly pipelined architecture to handle a 4096-size NTT task. Since the 2^n NTT sub-tasks with 2^m inputs do not have any data dependencies, each sub-task can reuse the m butterfly operation units to achieve the pipeline between sub-tasks. Fortunately, the latencies of butterfly operations across rounds are same, which makes it possible to pipeline two adjacent sub-tasks at the same stage. This pipeline is shown on the top half of Fig. 8. The inputs of each sub-task are alternately transported to the butterfly operation unit with a step size of 2^{m-1} in order, and the adjacent butterfly operation units can be executed continuously. Thus, the pipeline among sub-tasks is achieved. Since the initiation interval of each sub-task is reduced to half, this pipelined mode takes half the time of the serial computing mode to complete all sub-tasks.

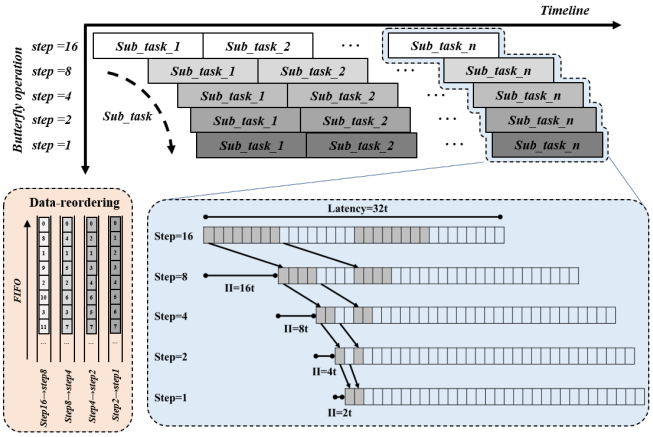


Fig. 8. Top half: two-level pipeline of NTT core. Bottom half: data-dependencies exist in 5 rounds of butterfly operations with different step sizes.

This two-level pipeline architecture achieves loop-level and function-level pipelines among multiple rounds of butterfly operations and multiple sub-tasks respectively. We implement it as a computation core shown in Fig. 9. The resource consumption of a pipeline can be easily adjusted by changing the values of n and m. In addition, we can further increase the number of concurrent sub-tasks that is we can instantiate more identical pipelines to improve the parallelism of sub-tasks. Manipulating the sub-task partitions and the number of pipelines, we can achieve specific speedup effects using different amounts of resources. Such high scalability makes this architecture especially friendly to FPGAs.

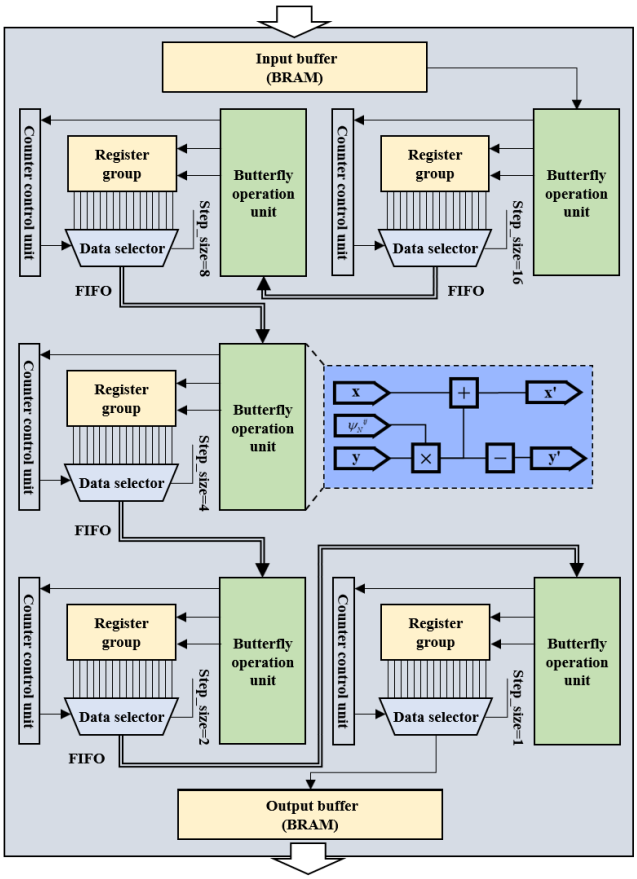


Fig. 9. NTT core with 5 rounds of butterfly operations.

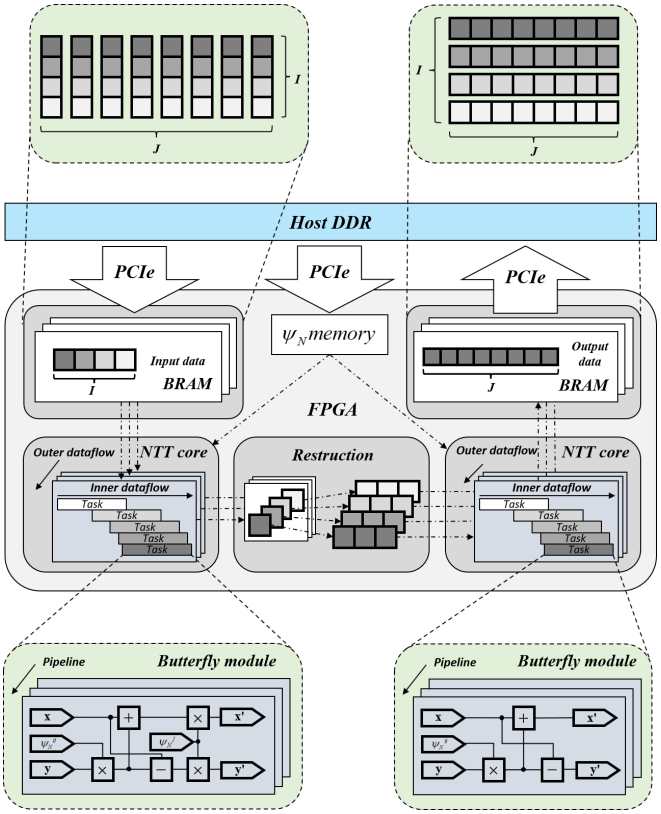


Fig. 10. The overall architecture of NTT computing.

D. Overall Architecture

We design a tightly pipelined architecture to process million-size NTT tasks, using the NTT core mentioned in III-C. The overall architecture is shown in Fig. 10. The N-size NTT task is divided into J I-size sub-tasks on the host side, and the inputs of these sub-tasks are streamed to the buffers on the FPGA board through the PCIe bus. These buffers are accessible from multiple ports. Once the data in the buffer is adequate to start the first round of butterfly operations in the NTT core, the first NTT core will be triggered, and the two-level pipeline will be filled gradually. Then, the NTT core generates the outputs of each sub-task continuously, and further those outputs will be rearranged through a set of FIFOs connected to the BRAM. The outputs of J I-size sub-tasks are distributed to J BRAM arrays using the data selector, which will regenerate I J-size sub-tasks. The inputs of new sub-tasks are transferred to another NTT core in turn, and a new NTT core will be triggered. Likewise, the two-level pipeline of the second NTT core will be filled up gradually, and the outputs of I sub-tasks will be generated sequentially. Finally, the outputs of the N-size NTT task flow out from the second NTT core, we collect and send the outputs back to the host side through the PCIe bus in batches.

It is worth mentioning that the twiddle factors are usually generated during the NTT calculation process. However, this generation consumes considerable DSP resources on an FPGA and degrades the computing performance. Therefore, we pre-compute all twiddle factors for a fixed-size NTT core, and store them off-chip, then transfer them to on-chip memory along with the input data before computation.

In addition, the divide-and-conquer calculation process of NTT requires multiplying the outputs of the sub-tasks, which are obtained in the first division of the task, by twiddle factors. To save additional hardware modules, we incorporate the above operation into the first NTT core. Therefore, each butterfly operation in the first NTT core is multiplied by twiddle factors twice, and it does not affect the performance of the entire architecture.

This two-level pipeline architecture has high scalability and can easily fit into an FPGA of different scales. As a result, we can effectively utilize FPGA resources to achieve maximum acceleration, by adjusting the size of the NTT core and partitioning the original NTT task properly.

IV. Result and Evaluation

A. Implementation and Evaluation Setup

We prototype the accelerator on AMD-Xilinx Alveo U50 (UltraScale+ XCU50 FPGA) using Vitis 2021.2 and implement the host side application on an up-to-date CPU, AMD Ryzen 9 5900X. A well-known zk-SNARK library, Bellman is used as the benchmark program to verify the correctness and

performance of our proposed NTT acceleration architecture. Proof-generation is implemented by the Groth16 algorithm, with 4 NTTs and 4 iNTTs. We store the inputs and outputs of NTT/iNTT in Bellman to the local as the benchmark for verifying the correctness of the calculation. At different scales, our kernel gets the same outputs as Bellman, using the same inputs.

We use C++ and Vitis HLS to implement the hardware acceleration kernel, including the top-level function and the underlying modules, such as the large-bitwidth modular arithmetic units. Then, we adopt the OpenCL to establish the communication with the hardware kernel for the host, managing the kernel calls and data transmission between the host and the kernel.

On the host side, since Bellman is developed in Rust, correspondingly we use the Rust-wrapped OpenCL library **opencl3** instead of the native OpenCL API. We have rewritten the interfaces of finding platforms and devices so that they can recognize AMD-Xilinx XRT and board devices. Further, we organize the NTT/iNTT tasks into a command queue, so that they can call the NTT kernel in sequence.

On the device side, we encapsulate the architecture proposed in III-C as a top function into a kernel, which is generated for a bit stream and burned on the board. In both NTT cores, on the one hand, we use the **pipeline** directive to implement a loop-level pipeline during iterations of butterfly operations with a fixed step size. On the other hand, the **dataflow** directive is used to implement the function-level pipelines among butterfly operations with different step sizes and sub-tasks. Fig. 9 shows where these two directives apply. In addition, we use HLS arbitrary-precision unsigned integer type **ap_uint** to define the large-bitwidth data types and arithmetic operations, which simplifies design and saves developing time.

B. Resource Consumption and Scalability

We implement the modular arithmetic units of 256 and 384 bits since most of the polynomial multiplications in zk-SNARK are performed on the finite fields of 256 and 384 bits. Table II provides a detailed resource consumption of large-bitwidth multiplication and Montgomery modular multiplication. We can easily use the **ap_uint** in HLS to generate an arbitrary bit-width multiplier through DSP. However, such direct generation costs too many DSPs and can hardly meet timing constraints. Therefore, we first use DSP and LUT to implement several small multipliers, and then cascade them into a larger multiplier, as described in III-A. This approach consumes a little more LUT resources (no more than 1% of the total on-chip) and slightly increases the latency (only a few clock cycles). However, as a return, the efficiency in the use of resources has been highly increased.

Table III provides a detailed resource consumption of a 4096-size NTT core, which is large enough to handle the sub-tasks divided from the million-size NTT task. Considering the size of NTT core is adjustable, we list the resource consumption on different butterfly operations cross rounds and different degrees of parallelism on sub-tasks. For example,

TABLE II
RESOURCE CONSUMPTION OF BASIC MODULES.

Module	Bit-width	DSP(%)	FF(%)	LUT(%)
Multiple	384	261(4.4)	7384(0.4)	11972(1.3)
	256	224(3.7)	1246(0.07)	1842(0.2)
Montgomery Multiple	384	261(4.4)	10468(0.6)	13839(1.6)
	256	224(3.7)	1540(0.08)	2017(0.2)

we can divide a large NTT task into 256 smaller sub-tasks. Compared to running all the sub-tasks in a single pipeline sequentially, the resource consumption is three times more, but the latency is three quarters less, if we evenly run those sub-tasks in 4 identical pipelines in parallel. The above rule still holds if we divide the same NTT task into 16 sub-tasks, or any arbitrary number of sub-tasks. In summary, regardless of how many rounds of butterfly operations a pipeline has, the overall latency of the NTT core decreases as the degree of the parallelism increases. In addition, the sub-task partition and the number of pipeline instances can be adjusted in an NTT core, making our accelerator highly scalable. Therefore, users can load an NTT core of different scales selectively according to the scale of available resources on the board, to achieve different degrees of acceleration.

TABLE III
RESOURCE CONSUMPTION OF NTT CORE.

DSP(%)	FF(%)	LUT(%)	BRAM(%)	Latency(clock)
8 rounds & 16 sub-tasks & 1 sub-task in parallel				
1795(30.2)	47329(2.7)	57058(6.6)	448(16.0)	39094
8 rounds & 16 sub-tasks & 2 sub-tasks in parallel				
3592(60.3)	101284(5.8)	118680(13.6)	986(35.1)	20361
4 rounds & 256 sub-tasks & 1 sub-task in parallel				
899(15.1)	28395(1.6)	33323(3.8)	320(11.4)	39407
4 rounds & 256 sub-tasks & 2 sub-tasks in parallel				
1798(30.2)	60822(3.5)	67699(7.8)	653(23.2)	20312
4 rounds & 256 sub-tasks & 4 sub-tasks in parallel				
3597(60.4)	130767(7.5)	137429(15.8)	1372(48.8)	10524

**The clock frequency is set to 68Mhz.*

C. Evaluating NTT Core against PipeZK

Although Kawamura et al. [37] and Ozcan et al. [38] also used HLS to implement hardware accelerators for NTT, they could only support 64-bit or 1024-size NTT tasks. In this section, we compare our NTT core with the POLY module in PipeZK, which is state-of-the-art work implemented under UMC 28nm library. In the POLY module, there is also a core that performs NTT calculations in a pipelined manner. To evaluate the performance of our architecture excluding the influence of external factors, such as development method and clock frequency, we replicate the NTT core of PipeZK on the U50 for comparison. Table V shows the latency and speedup of the two architectures at different input sizes and data bit-widths. In addition, table VI shows the throughput comparison of the two architectures. We define the computation throughput of NTT as the ratio of the input data volume to the processing time.

TABLE IV
RESOURCE COMPARISON OF NTT CORE.

Work	DSP(%)	FF(%)	LUT(%)	BRAM(%)
Ours	1795(30.2)	47329(2.7)	57058(6.6)	448(16.0)
PipeZK	1795(30.2)	35496(2.0)	45075(5.2)	286(10.4)

**Implemented with one pipeline consist of 8 butterfly operation units.*

TABLE V
LATENCY(IN MILLISECOND) COMPARISON OF NTT CORE.

Size	Bit width			
	256-bits		384-bits	
	PipeZK	Our work	PipeZK	Our work
4096	6.772	3.984(1.7×)	14.424	8.327(1.73×)
8192	13.554	7.737(1.75×)	29.819	16.866(1.76×)
16384	28.446	15.205(1.87×)	64.572	34.363(1.88×)
32768	55.469	29.346(1.89×)	129.797	67.789(1.91×)
65536	114.455	59.722(1.91×)	263.696	137.597(1.91×)
262144	468.860	240.973(1.94×)	1064.406	553.029(1.92×)
1048576	1892.553	968.722(1.95×)	4433.208	2294.065(1.93×)
2097152	4039.993	2067.869(1.95×)	9106.756	4694.532(1.94×)

**Implemented at the clock frequency of 68Mhz.*

In PipeZK, the latency and initiation interval of butterfly operations in a round are both $2T$. While in our NTT core, with the help of the pipeline design among multiple sub-tasks, the initiation interval is reduced to T. As a result, the overall latency is reduced from $2nT$ to nT. The results in Table V show that our NTT core gets better performance improvement as the input size increases. Once the data flow among sub-tasks is large enough, the speedup is very close to twice, the theoretical maximum speedup. In addition, throughput has almost doubled. Importantly, compared to PipeZK, table IV shows that our NTT core consumes very limited additional resources.

D. Evaluation with Workloads

1) Performance: Since the roles asking for proof-generation are a large number of discrete individuals, the computation latency is the key to evaluate the performance. In addition, when zk-SNARK is deployed in real scenarios, energy efficiency is another key indicator to evaluate service quality. We load our NTT accelerator into Bellman with NTT tasks and evaluate it on U50 against on CPU. Table VII shows the computation latency, energy efficiency, and throughput of both.

TABLE VI
THROUGHPUT(IN MB/S) COMPARISON OF NTT CORE.

Size	Bit width			
	256-bits		384-bits	
	PipeZK	Our work	PipeZK	Our work
4096	18.458	31.375(1.7×)	12.999	22.517(1.73×)
8192	18.444	32.312(1.75×)	12.575	22.234(1.76×)
16384	17.577	32.883(1.87×)	11.614	21.825(1.87×)
32768	18.028	34.076(1.89×)	11.556	22.127(1.91×)
65536	17.474	33.488(1.91×)	11.376	21.802(1.91×)
262144	17.062	33.198(1.94×)	11.273	21.698(1.92×)
1048576	16.908	33.033(1.95×)	10.827	20.923(1.93×)
2097152	15.841	30.949(1.95×)	10.541	20.449(1.94×)

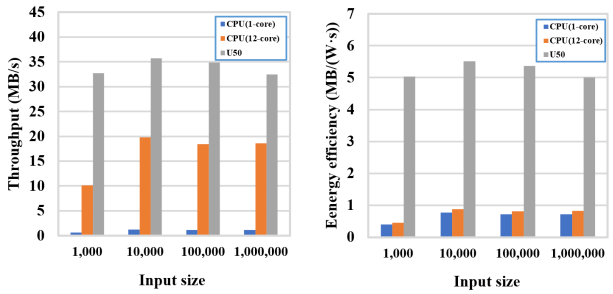
**Implemented at the clock frequency of 68Mhz.*

TABLE VII
PERFORMANCE EVALUATION OF SINGLE NTT TASK UNDER BELLMAN WORKLOAD.

Input size	Indicator	1-core	12-core	U50
1,000	Perf.	103.652	6.497	2.006
	(ms)	51.671×	3.239×	1×
	Throughput	0.632	10.087	32.669
	(MB/s)	51.691×	3.239×	1×
	Energy efficiency	0.395	0.449	5.035
	(MB/(W·s))	12.747×	11.214×	1×
10,000	Perf.	847.924	52.964	29.346
	(ms)	28.894×	1.805×	1×
	Throughput	1.237	19.799	35.731
	(MB/s)	28.885×	1.805×	1×
	Energy efficiency	0.773	0.881	5.506
	(MB/(W·s))	7.123×	6.250×	1×
100,000	Perf.	7295.126	455.822	240.937
	(ms)	30.278×	1.892×	1×
	Throughput	1.149	18.403	34.817
	(MB/s)	30.302×	1.892×	1×
	Energy efficiency	0.718	0.819	5.366
	(MB/(W·s))	7.474×	6.552×	1×
1,000,000	Perf.	57861.36	3616.648	2067.869
	(ms)	27.981×	1.749×	1×
	Throughput	1.160	18.632	32.453
	(MB/s)	27.977×	1.742×	1×
	Energy efficiency	0.725	0.829	5.001
	(MB/(W·s))	6.898×	6.033×	1×

As the only difference between NTT and iNTT is the value of the twiddle factors, we just experiment with NTT tasks in this section. Therefore, we test the performance of a single NTT task in Bellman under AMD Ryzen 9 5900X(3.7GHz) with a single core and 12 cores with 24 threads as benchmarks. Accordingly, we evaluate various performance indicators of the same NTT task on Alveo U50. Computation energy efficiency refers to the ratio of computation throughput to computation power consumption. When NTT tasks are executed on the CPU, the computation latency of 12 cores is about 16 times that of a single core. However, our NTT accelerator still has a speedup of about 1.75 times compared to 12 cores at the million-size, and the computation throughput is nearly 32MB/s, 1.74 times that of 12 cores. We use Powertop and Vitis analyzer to measure the NTT core's power on CPU and FPGAs, respectively, and calculate the energy efficiency. Compared to a CPU with 12 cores, our NTT accelerator can reach 6 to 7 times energy efficiency. Fig. 11 demonstrates the significant advantages of our accelerator in terms of throughput and energy efficiency.

2) Computation Stability: We perform 100 million-size NTT tasks on the CPU and U50 respectively, record the running time, and plot a graph shown in Fig. 12. The performance is more stable on U50 than on the CPU, due to that on U50, the variance of calculation time of NTT module is much smaller, and the jitter is more subtle.



(a) Computation throughput (b) Computation energy efficiency

Fig. 11. Computation throughput and energy efficiency of NTT under U50 and CPU.

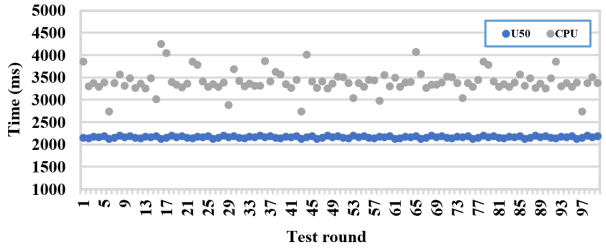


Fig. 12. The computation stability of NTT between U50 and CPU.

V. Conclusion

In this paper, we propose an FPGA-based hardware architecture for NTT in zk-SNARK. An energy-efficient accelerator with a tight pipeline is implemented, using the proposed "data-reordering" technique. In our accelerator, the rounds of butterfly operations, the sub-task partitioning, and the degree of parallelism can be adjusted according to the on-chip resources, ensuring its high scalability. As a proof-of-concept, the accelerator outperforms state-of-the-art accelerators and zk-SNARK libraries in terms of speed and energy efficiency. Moreover, our work shows that combining the advantages of OpenCL, HLS and scalable hardware design will be a promising choice for ZKP's high-efficiency computing. For further improvement, we have two future directions, including building up efficient buffers to well balance the performance between the NTT core and I/O communication, and making multiple NTT cores run continuously across calls.

VI. Acknowledgement

This work was supported by the National Natural Science Foundation of China (61972180) and the Project of Jiangsu Provincial Engi-neering Laboratory of Pattern Recognition and Computational Intelligence. The corresponding authors are: Zhilei Chai (zlchai@jiangnan.edu.cn).

VII. Conflict of Interest

Authors have no conflict of interest relevant to this article.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[2] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning shabby x. 509 certificates into elegant anonymous credentials with the magic of verifiable computation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 235–254.

[3] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio coin: building zerocoin from a succinct pairing-based proof system," in *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*, 2013, pp. 27–30.

[4] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[5] H.-S. Huang, T.-S. Chang, and J.-Y. Wu, "A secure file sharing system based on ipfs and blockchain," in *Proceedings of the 2020 2nd International Electronics Communication Conference*, 2020, pp. 96–100.

[6] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 2012, pp. 326–349.

[7] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer, "The hunting of the snark," *Journal of Cryptology*, vol. 30, no. 4, pp. 989–1066, 2017.

[8] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 329–349.

[9] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.

[10] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 626–645.

[11] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vsql: Verifying arbitrary sql queries over dynamic outsourced databases," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 863–880.

[12] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[13] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 416–428.

[14] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks," in *Annual International Cryptology Conference*. Springer, 2017, pp. 581–612.

[15] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[16] S. Bowe and A. Gabizon, "Making groth's zk-snark simulation extractable in the random oracle model," *Cryptology ePrint Archive*, 2018.

[17] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Annual International Cryptology Conference*. Springer, 2019, pp. 733–764.

[18] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2111–2128.

[19] E. Haleplidis, T. Tsakoulis, A. El-Kady, C. Dimopoulos, O. Koufopavlou, and A. P. Fournaris, "Studying opencl-based number theoretic transform for heterogeneous platforms," in *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2021, pp. 339–346.

[20] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 56–64.

[21] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, 2016.

[22] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, 2014.

[23] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner, "Efficient fpga implementations of lowmc and picnic," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 417–441.

[24] R. Agrawal, L. Bu, A. Ehret, and M. Kinsy, "Open-source fpga implementation of post-quantum cryptographic hardware primitives," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 211–217.

[25] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2019, pp. 253–260.

[26] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[27] "bellperson: Gpu parallel acceleration for zk-snark," https://github.com/filecoin-project/bellperson, 2020.

[28] "bellman: zk-snark library," https://github.com/zkcrypto/bellman, 2018.

[29] G. Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography," *Cryptology ePrint Archive*, 2018.

[30] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[31] E. Öztürk, "Modular multiplication algorithm suitable for low-latency circuit implementations," *Cryptology ePrint Archive*, 2019.

[32] A. Karatsuba, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, pp. 595–596.

[33] G. C. Chow, K. Eguro, W. Luk, and P. Leong, "A karatsuba-based montgomery multiplier," in *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 2010, pp. 434–437.

[34] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[35] T.-W. Sze, "Schönhage-strassen algorithm with mapreduce for multiplying terabit integers," in *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*, 2012, pp. 54–62.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[37] K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast number theoretic transform," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2018, pp. 106–111.

[38] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, vol. 12, no. 4, pp. 133–136, 2019.