Hardware Acceleration and Implementation of Fully Homomorphic Encryption Over the Torus

Tianqi Kong and Shuguo Li, *Member, IEEE*

***Abstract*— Fully Homomorphic Encryption (FHE) allows direct computation on ciphertext without decryption which provides an effectual approach for privacy-preserving computation. Bootstrapping is an useful feature to have in FHE scheme. However, limited by substantial memory requirements and costly computations, it is difficult to put bootstrapping into practice. It's necessary to implement hardware acceleration for bootstrapping. In this paper, we proposed the first FPGA implementation of TFHE bootstrapping with 128-bit security to our acknowledgement. We designed a fixed distribution access pattern to accelerate polynomial multiplication and the simplified modular reduction circuit is designed to optimize the computations. The high-level pipelines are utilized to reduce the clock cycle consumption. We proposed multiple parallelism-matching architecture to further speed up the bootstrapping procedure. We implement entire bootstrapping scheme with various key unrolling factors on FPGA platform and synthesis an individual computational core in TSMC 28nm for performance evaluation. The experimental results indicate that our bootstrapping scheme can achieve 5.5 × −34× speed up compared with the state-of-art CPU baselines. Compared with previous FPGA design, our acceleration achieves a latency improvement of 6% − 65% with higher security.**

***Index Terms*— Fully homomorphic encryption, bootstrapping, TFHE, polynomial multiplication, FPGA.**

I. Introduction

THE theory of Fully Homomorphic Encryption (FHE) provides an effectual approach for privacy-preserving computation [1]. Recent developments in cloud service have heightened the need for data security and privacy. FHE allows direct computation on ciphertext without decryption. Users can locally upload the encrypted data to cloud server without security concern, the cloud servers perform computations on ciphertext, and then return the result, which is still encrypted. All computations on cloud are performed in the ciphertext space.

However, each computation on ciphertext introduces noise. When accumulated noise exceed the threshold, the ciphertext cannot be decrypted correctly. One of the solutions is the Somewhat Homomorphic Encrypyion (SHE) [1], [2], [3], [4], but homomorphic computations in SHE scheme have limited depth. To overcome the noise obstacle, Gentry proposed bootstrapping [1], which homomorphically evaluates the decryption circuit on ciphertext to "refresh" the ciphertext noise level so that the noise is kept at a fixed level.

Recent studies in the field of GSW-based schemes [5], [6], [7], [8], [9], [10] have optimized the bootstrapping procedure. The Fully Homomorphic Encryption over the Torus (TFHE) scheme can complete a gate-bootstrapping within 0.1 seconds [7]. Several software attempts have been made to accelerate TFHE-bootstrapping as [14] on CPU, [28] on GPU. However, limited by complexity computation, the performance of software implementations is difficult to realize practical TFHE-based computing. It's necessary to implement hardware acceleration for TFHE. Reference [11] implemented an FPGA acceleration with various key unrolling factors, but the parameter sets in [11] only match 80/110-bit security. In addition, it has a high memory consumption with only 32-bit NTT-prime; [12] accelerated the polynomial multiplication by adopted Fast Fourier Transform (FFT) [17]. Because of the approximate float-point FFT, the computations in scheme have suffered from accuracy loss. The hardware design in [12] does not have any pipelined design or optimization, it needs to be further optimized; Almost all hardware designs suffered from the substantial memory requirements and costly computations. In this paper, we propose an efficient hardware implementation of TFHE-bootstrapping with 128-bit security parameter set. To our acknowledgement, this is the first hardware implementation with 128-bit security. We focus on the architecture and parallelism-memory tradeoff with in-depth analysis of TFHE scheme. We implement the entire TFHE bootstrapping on the Virtex UltraScale+ VCU128 FPGA platform, including NTT core. Meanwhile, an individual NTT core is designed and implemented in ASIC for performance evaluation. The contributions are as follows:

- The overall architecture of 128-bit security TFHE bootstrapping hardware scheme with multiple parallelism-matching;
- The high-level pipelined parallel NTT module is designed with fixed distribution unit to accelerate the polynomial multiplication. Our NTT module have at least 30% improvement in ATP (Area-Delay Product) with larger coefficient width;
- The simplified modular reduction unit is proposed to avoid the multiplication, further speed up the computations;

Manuscript received 18 July 2023; revised 11 October 2023; accepted 29 November 2023. Date of publication 13 December 2023; date of current version 28 February 2024. This work was supported by the National Natural Science Foundation of China under Grant 61974083. This article was recommended by Associate Editor C. H. Chang. *(Corresponding author: Shuguo Li.)*

The authors are with the Institute of Microelectronics, Tsinghua University, Beijing 100084, China (e-mail: ktq18@mails.tsinghua.edu.cn; lisg@tsinghua.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2023.3338953.

Digital Object Identifier 10.1109/TCSI.2023.3338953

- The on-chip key switching module is implemented with reduced memory requirements and computational complexity;
- The implementations with various key unrolling factors utilized the wasted space to save the memory consumption. Our hardware scheme with higher security achieves a latency improvement of $6-65\%$ compared with state-of-art FPGA designs.

The paper has been organised in the following way: Section II gives an introduction to TFHE gate bootstrapping and reviews previous related researches; Section III proposes our hardware TFHE-bootstrapping architecture and describes the methodology of every functional module optimization; The experimental results and performance comparisons with previous works are presented in Section IV; Conclusions follow in Section V.

II. Background

Notation: In the rest of the paper, we denote the integers module q as $\mathbb{Z}_q$. $\mathbb{B}$ is the representation of set{0,1}. The Torus of real numbers modulo 1 are represented by $\mathbb{T}$. The polynomials with coefficients in $\mathcal{R}$ are denoted as $\mathbb{R}[X]$, and $\mathbb{R}_N[X]$ means $\mathbb{R}[X]/(X^N+1)$. Same as $\mathbb{R}_N[X]$, $\mathbb{Z}_N[X]$ and $\mathbb{T}_N[X]$ mean polynomial which coefficients in $\mathbb{Z}$, $\mathbb{T}$, respectively. $\mathcal{M}_{p,q}(E)$ denotes a p×q matrix with elements in E. We use n to represent the dimension of TLWE cipher. N is used to represent the degree of polynomial as $\mathbb{R}[X]/(X^N+1)$ shows.

A. TFHE Bootstrapping

Bootstrapping, as a method to reduce the ciphertext noise, is a useful feature in FHE scheme. Compared with other FHE schemes, on one hand, the significant strength of TFHE scheme is the efficient bootstrapping: The bootstrapping operation for BGV, BFV, CKKS take hundred seconds, but TFHE can complete a bootstrapping operation within 13 ms. On the other hand, each computation on ciphertext introduces noise. When accumulated noise exceed the threshold, the ciphertext cannot be decrypted correctly. Bootstrapping operation can decrease the ciphertext noise and keep it at a fixed level. Due to the extremely expensive bootstrapping cost, some schemes (eg. BGV, BFV, CKKS) only support limited computational depth by designing a large enough noise threshold. However, the efficient bootstrapping procedure of TFHE can be executed after every homomorphic computation, making it feasible to compute homomorphic computations with arbitrary depth. Finally, the non-linear functions supported by TFHE are suitable for applications: Other schemes like BGV only support homomorphic additions and multiplications, the widely used non-linear calculation are not supported. TFHE scheme homomorphically evaluates logic gates, which can construct arbitrary computations and thus makes it possible to support arbitrary operations.

TFHE homomorphically map $c \in \mathbb{T}$ to $f(c) \in \mathbb{Z}_N[X]$, thus the decryption of LWE cipher c can be constructed correspondingly by homomorphic accumulator. The procedure of TFHE-Bootstrapping is shown in Algorithm 1.

Algorithm 1 TFHE Bootstrapping Procedure

Input: LWE cipher $c_{s,\eta}=(\mathbf{a}, b) \in \mathbb{T}^{n+1}$, secret key $s \in \mathbb{B}^n$
bootstrapping key $BK_{s\to s'',i=(1,n)} \in \mathbb{T}_{q,N}[X]^{(k+1)l\times(k+1)}$,
a keyswitch key $KS_{s'\to s,i=(1,N),j=(1,t)} \in \mathbb{T}^{n+1}$
two fixed messages $\mu_0, \mu_1 \in \mathbb{T}$
Output: LWE cipher $c'_s(\mu_0 \to \varphi_s(\mathbf{a}, b) \in \left]-\frac{1}{4}, \frac{1}{4}\right[; \mu_1 else)$

1: Let $\overline{\mu} = \frac{\mu_0+\mu_1}{2}$ and $\overline{\mu}' = \mu_0 - \overline{\mu}$
2: Let $\overline{b} = \lfloor 2Nb \rceil$ and $\overline{a_i} = \lfloor 2Na_i \rceil$ for each $i \in [1, n]$
3: Let $testv := (1+X+\ldots+X^{N-1}) \times X^{-\frac{2N}{4}} \cdot \overline{\mu}' \in \mathbb{T}_N[X]$
4: $ACC \leftarrow (X^{\overline{b}} \cdot (\mathbf{0}, testv)) \in \mathbb{T}_N[X]^{k+1}$
5: **for** $i = 1$ **to** n:
6: $\quad ACC \leftarrow \left[\mathbf{h} + (X^{-\overline{a_i}} - 1) \cdot BK_i\right] \boxdot ACC$
7: Let $\mathbf{u} := (\mathbf{0}, \overline{\mu}) + SampleExtract(ACC)$
8: $SampleExtract(ACC) \leftarrow u'$:
9: $\quad ACC \leftarrow (a'_0, a'_1, \ldots, a'_{N-1}), (b'_0, b'_1, \ldots, b'_{N-1})$
10: $\quad u' \leftarrow (a'_0, -a'_{N-1}, -a'_{N-2} \ldots, -a'_1, b'_0)$
11: Return $KeySwitch_{KS}(\mathbf{u})^*$

$*$: The algorithm of KeySwitch is detailed in Algorithm 4, Page 8.

Analyze the bootstrapping procedure, it can be seen that the serial "BlindRotate" step (line 5,6) determines the execution time. It is also worth noting that, the size of bootstrapping key involved in polynomial multiplication increases with security parameter. The frequently memory access also affects the performance of bootstrapping computation.

B. Related Works

Recently, researchers have implementated various bootstrapping schemes based on TFHE. The software implementation [29] in C++ can achievex 13ms per gate on 64-bit single core (intel i7-4910MQ) at 2.90 GHz with generic fast FFT library FFTW [9]. CuFHE-bootstrapping [28] is measured on NVIDIA GeForce RTX 3090 GPU @1.70 GHz. Benefits from fast polynomial multiplication based on NTT, it can complete bootstrapping in 9.34 ms [11]. However, due to the great amount of complex computation and frequent external memory access, it is difficult for software to put TFHE bootstrapping into practice.

Compared with software platform, the hardware platform has the characteristics of high parallelism. It's suitable for the implementation of bootstrapping algorithm. The on-chip memory of hardware design can reduce the latency of memory access thus further optimize the bootstrapping performance. There have been several hardware implementations for TFHE bootstrapping in recent years. MATCHA [12] implemented TFHE-bootstrapping in ASIC with 16nm PTM process. This implementation accelerate polynomial multiplication through FFT. Due to the approximate floating-point arithmetic in FFT, accuracy loss will be introduced in polynomial computation. The hardware design in [12] does not have any pipelined design or optimization which needs to be further optimized; [11] implemented a FPGA acceleration with high memory consumption, in which the selected NTT prime only 32-bit. The parameter sets selected by [11] only provided 80/110-bit security; The other FPGA implementation [13] does not utilize

TABLE I
CLOCK CYCLE CONSUMPTION OF BOOTSTRAPPING STEPS

Step	Sub-step	Clk Consumption
Initialization		n
Plaintext Rotate		N
Blind Rotate(*n)	$(x^a-1)*bk$	$(k+1)N$
	$decompose$	$(k+1)Nl$
	$matrix$	$(k+1)^2lN^2$
	$accumulate$	$(k+1)^2lN$
Extract		$Nk+1$
Keyswitch		$(Nk+1)t$

neither pipeline nor other parallel structure. It takes extremely long time to compute bootstrapping.

In this paper, we proposed a multiple parallelism-matching hardware design with 128-bit security on FPGA to accelerate bootstrapping. The implementation utilize simplified memory access NTT and further accelerate polynomial multiplication via high-level pipelined design. The functional modules in hardware scheme like "Modular reduction" are optimized according to the resource consumption and computation characteristics individually. Considering the selected hardware platform, comparative experiments are carried out for various key unrolling factors. In the proposed hardware implementation, the on-chip key switching is designed to reduce the memory requirement and data transfer. The details of our hardware design are shown in next section.

III. BOOTSTRAPPING HARDWARE ARCHITECTURE

The TFHE bootstrapping algorithm can be divided into the following steps: "Initialization", "PlainRotate", "BlindRotate", "Extract", "Key Switching". For hardware implementation of bootstrapping algorithm, the clock cycles consumed by each step are shown in Table I. In Table I, a is an element of input cipher, n is the TLWE cipher dimension and N is the degree of polynomial. k is an integer usually taken as 1 or 2 in parameter set and l is a decomposition parameter. t is a precision parameter in key switching procedure.

The computations in Table I correspond to the related steps in Algorithm 1, designed to functional modules in bootstrapping hardware implementation. In this section, we present our hardware design with detailed description of each functional units. To obtaining more efficient hardware bootstrapping scheme, the functional modules are optimized according to resource consumption and computation characteristics.

A. Design of NTT/INTT Acceleration

In TFHE bootstrapping, the serial "BlindRotate" loops consume a great part of latency as Table I shows. The most expensive sub-step in "BlindRotate" is "Matrix" computation, its clock cycle consumption mainly comes from the polynomial multiplication over ring, which has a computational complexity of $O(N^2)$. Polynomial multiplication is also the most complex computation in entire bootstrapping procedure. The clock cycles consumed by serial accumulated polynomial multiplication are considerable among all functional modules in bootstrapping computation. In this part, we mainly describe the optimization methods for polynomial multiplication module.

Algorithm 2 Polynomial Multiplication Based on NTT

Input: polynomials $A, B \in Z_q[X]$ of degree $N-1$, and $2N$th primitive root $\omega_{2N} \in Z_q$ of unity
Output: product polynomial $C \leftarrow A \times B$ of degree $N-1$

for $(m = 1; m < n; m = 2m)$:
  for $(i = 0; i < m; i++)$:
    for $(j = \frac{i \cdot N}{m}; j < \frac{(2j+1)N}{2m}; j++)$:
      $v = MultR(a_{j+\frac{n}{m}}, a_j, \omega_{2N}^{i}, p)$
      $a_j = a_j + v \ (modp)$
      $a_{j+\frac{n}{m}} = a_j - v \ (modp)$ $\triangle BFU$
      $same\ to\ b_j, b_{j+\frac{n}{m}} \ldots$
$\widetilde{A} \leftarrow A, \ \widetilde{B} \leftarrow B$
$\widetilde{C} \leftarrow digitwiseMult(\widetilde{A} \cdot \widetilde{B})$
for $(m = n/2; m < 1; m = m/2)$:
  for $(i = 0; i < m; i++)$:
    for $(j = \frac{i \cdot N}{m}; j < \frac{(2j+1)N}{2m}; j++)$:
      $v = (c_j - c_{j+\frac{n}{m}})/2 \ (modp)$
      $c_j = (c_j + c_{j+\frac{n}{m}})/2 \ (modp)$
      $c_{j+\frac{n}{m}} = MultR(v, \omega_{2N}^{-i}, p)$ $\triangle BFU$
$C \leftarrow \widetilde{C}$
Return polynomial product C

1) Number Theoretic Transform (NTT): Polynomial multiplication is commonly accelerated by Fast Fourier Transform (FFT) algorithm [17], Tom-cook multiplication [18], KO multiplication [19]. Among these algorithms, FFT has the lowest complexity $O(NlogN)$, which is suitable for polynomial multiplication with large degree. Considering the precision, the designer of TFHE uses 64-bit double-precision floating-point number for FFT transform [29]. However, we carried out experiments for Floating-Point Unit (FPU) and integer multipliers, the 52-bit multiplier we used saves about half of the resources than 64-bit double-precision FPU. Considering for parallel design, the consumption of FPU is obviously expensive. As an integer generalization of FFT, NTT only performs computations over the integers is well-suited for the modern FPGA devices, the DSP resources of which are integer-based. The precise NTT transform will not introduce additional computational noise in bootstrapping procedure.

In order to avoid the "zero padding" in NTT and polynomial reduction, the polynomial multiplication over $Z[X]/(X^N+1)$ is generally optimized by negative-wrapped convolution NTT as Algorithm 2 [15]. In NTT-based polynomial multiplication, firstly, two polynomials are computed to NTT-domain by NTT transform; Then, the digit-wise multiplication of NTT-domain coefficients are carried out; Finally, the INTT transform is executed to converts the NTT-domain coefficients back to the original domain as the product polynomial coefficients.

As Algorithm 1 shows, the two operands of polynomial multiplication in bootstrapping are test vector ACC and bootstrapping key BK_i, respectively. Compared with loading the original bootstrapping key BK_i, we directly load NTT-domain polynomial coefficients into the multiplication module. Thus,

half of the computation resource can be saved in "Matrix" polynomial multiplication process, and it is beneficial to further improve the computational parallelism inside the design. The twiddle factors in line 4,15 of Algorithm 2 are also fixed when the prime modulus q of NTT is selected. Hence, the fixed twiddle factors are stored in ROM table with lower computational cost. Due to the order-reversed property in NTT/INTT transform, the polynomial multiplication module adopts Cooley-Tukey flow for NTTs and the Gentlemen-Sande flow is selected for INTTs to avoid the bit-reverse process in NTT/INTT transform.

2) Parallel Architecture of NTT/INTT: NTT-based polynomial multiplication has the reduced complexity of $O(NlogN)$, but even so, the clock overhead of it will still be accumulated n times by serial "BlindRotate" process. For example, if the latency of a single polynomial multiplication is $5us$, the latency consumed by polynomial multiplication in bootstrapping will be $3ms$. The polynomial multiplication obviously needs to be further accelerated.

The unit that computes line 4-6/13-15 in Algorithm 2 is called butterfly unit (BFU). In each round of the NTT transform, there are $N/2$ numbers of butterfly computation will be performed. The inputs of each BFU in each round (polynomial coefficients) are different and independent. In order to reduce the clock cycle cost of NTT/INTT and further accelerate polynomial multiplication, our design parallelizes the data-independent butterfly computations. The NTT, INTT transform modules are designed with multiple BFUs for parallel computing as shown in Fig.1(a).

The parallelism of NTT/INTT module is designed considering the hardware platform and the cost of implementation resources. If the parallelism of each unit is set different, the data bus between units needs bandwidth conversion, which consumes additional resources and execution time. Bandwidth conversion is a needed step for handshake between two modules with different data bus widths. In hardware design, the different computations between modules, the difference of memory units inside modules (in FPGA, usually RAM IP), and the different parallelism in modules, will lead to different bandwidth of modules. The bandwidth is presented as data width and number of data in/out wires. In handshake for data transfer between two modules, if the output module with smaller data width (less data wires) and the receiver module with greater data width (more data wires), the issue of dissimilar bandwidth is not important apart from wasting some of the memory. However, if the output module with greater data width (more data wires) and the receiver module with smaller data width (less data wires), additional data repeater is required because of the bandwidth deficiency of the receiver module. This is the bandwidth conversion we mentioned. Bandwidth conversion not only wastes the memory of data repeater, but more importantly, it also takes a great deal of clock cycles to re-load and re-send the data. This is an large non-essential overhead for the system clock. The clock overhead of bandwidth conversion is enormous and needless in hardware design.

Therefore, the related modules in serial "BlindRotate" step are set to same parallelism (such as digit-wise multiplication module between NTT and INTT) to avoid bandwidth conversion.

Before NTT parallelism selection, it is necessary to clarify a point that increasing the number of BFUs can not results in a stable latency improvement. Firstly, since the polynomial multiplication consumes chief clock cycles in bootstrapping procedure, we set NTT as the center of optimization. The optimization focus on polynomial multiplication can bring considerable benefits to the bootstrapping scheme.

Secondly, in order to avoid the unnecessary clock cost introduced by bandwidth conversion, the related modules in serial "BlindRotate" step are set to same parallelism. The increase of BFU numbers not only increases the complexity of NTT module, but also leads to the increase of parallelism of other modules due to parallelism-matching. The almost globally increase of parallelism results in larger resource consumption and more congested routing. The computational resources, memory and routing resources on FPGA are all restricted. The large consumption affects the optimization space for other functional modules. The utilization rate of resources also affects the design frequency to a certain extent.

The last but not least factor is routing. (a) BFU in NTT perform crossover computation in the $u + v, u - v$ format. Such crossover computation produces crossed wires in the placement and routing stages. Even if without parallel BFUs, the NTT unit also consumes a great deal of routing resources. (b) In our NTT module, we utilize multi-BFUs to further accelerate computations. In this case, the coefficients access pattern has high complexity, so that leads to a routing pattern with high complexity, which causes routing congestion, and in turn, affecting the timing closure. (c) The routing resources are already tight because of NTT with multi-BFUs. Moreover, larger parallelism leads to more complex control logics. It is worth noting that, the fan-out of control logics is high due to the large parallelism, which need more copied registers to ensure the drive capability. This results in more tight routing resources and more severe congestion, which increases the net delay. These three aspects make the routing congestion in scheme become an important factor affecting the timing closure.

Considering the global bootstrapping performance in parallelism-resource tradoff condition, after several experiments of various parallelism, we selected 8 BFU parallel NTT structure.

3) Simplified Memory Access: There are two methods for NTT memory access: in-place computation and not-in-place computation. In in-palce NTT transform, the coefficients will still be written back to the original read address after BFU computation. For not-in-place NTT, the fixed input structure is commonly used. Such structure has the fixed datapath from memory to the input ports of BFU, but after the calculation, the write-back address is different from the original read address. In hardware implementation, the different read/write addresses of non-in-place NTT will bring potential read-write conflicts to memory access. The ping-pong structure of memory can avoid the read/write conflicts, however, such ping-pong structure will at least double the memory consumption, which is obviously expensive. In addition, there is only a fixed read-input data

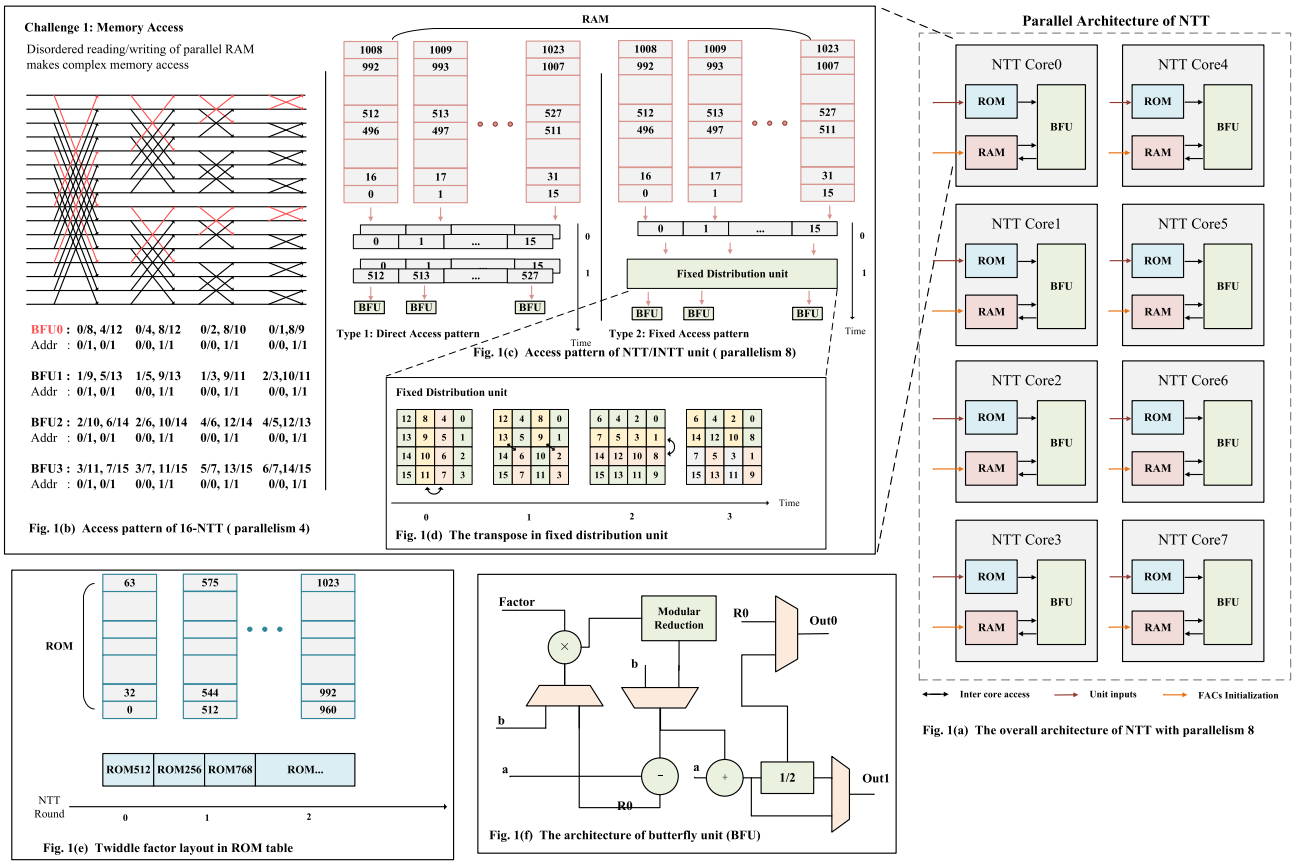

Fig. 1. The overall architecture of NTT unit with parallelism 8 and the design of simplified memory access pattern.

path in not-in-place NTT, the write-back structure still changes with NTT rounds, which still needs to be designed specifically. Therefore, the in-place NTT is selected for our polynomial multiplication module.

As shown in Algorithm 2 line 1-3/10-12, the memory access of NTT/INTT transform is related to computation round m, computation step i and the interval. In Fig. 1(b), we take 16-point NTT with parallelism 4 as an example to show the access pattern of in-place NTT. The red lines in figure represent the access pattern of BFU0, and the "Addr" represents the address of coefficients in parallel BlockRAM. It can be seen from Fig. 1(b) that, for NTT with parallel structure, the input and write-back order of each BFU varies with the computation, and the reading and writing patterns are disordered, making memory access of NTT complex. In iterative NTT/INTT, the logic distribution unit must be designed to meet the requirements of all rounds and times, which is a design bottleneck in parallel NTT/INTT.

We build the address generation unit and read the polynomial coefficients from RAMs in corresponding address. These coefficients will be distributed to the corresponding BFU ports through the logic distribution unit. As Fig. 1(b), the coefficients in RAM are stored in parallel under the consideration of data fan-out. As described in the above section, our NTT module is equipped with 8 BFUs for parallel transformation. In this case, 16 polynomial coefficients should be read from RAM in a single cycle. Therefore, We set 16 parallel Block RAMs to storage the polynomial coefficients.

For the coefficients distribution to BFU ports, the enumeration distribution is commonly used as a simple method. Enumeration distribution lists all access pattern of each transform round, and uses a large multiplexer to select the different pattern for different rounds. The architecture of the enumeration distribution is highly complex. If num BFUs are used, there will be $\log N \cdot 2num \cdot 2num$ connections. Such a high-degree MUX and the complex connection congest the route, which impacts the timing of its closure, limits the optimization space for other functional modules.

Observing the input variation of each BFU in NTT transform, we find that all computation round can be divided into two types. In $0 \sim \log N - \log num - 2$ rounds, two coefficients of a single BFU are stored in different addresses in parallel RAM. As Type 1 in Fig. 1(c), after reading the coefficients from RAM, the distribution unit only needs naturally input the coefficients to BFU ports. By naturally doing this, the access pattern of Type 1 can be constructed correctly. In this type, the access pattern is fixed under the various RAM address, and structure of Type 1 distribution unit is simple.

However, in Type 2 transform rounds ($\log N - \log num - 1 \sim logN - 1$), the two BFU input coefficients are stored at the same address, and the specific position in parallel RAM varies according to the computation times and rounds. By observing the regulation of access pattern, we proposed

a fixed distribution unit, which can be generally adopted to every rounds in Type 2 as Fig. 1(c) shows. This distribution unit in Fig. 1(d) only needs a maximum of three coefficient switches to complete the correct access pattern for each round in Type 2. Compared with the enumeration distribution of $\log N \cdot 2num \cdot 2num$, this fixed distribution unit significantly reduces the structural complexity of NTT memory access and saves the circuit route resources. In comparative experiments, the enumeration structure (NTT parallelism 2) can only be routed at 150MHz with high-level routing congestion, whereas the fixed-distribution unit method in this paper can be successfully routed at 180MHz with NTT parallelism 8. The simplified NTT access pattern by fixed distribution units improves the frequency of hardware design and helps optimise other functional modules in the implementation.

After the parameter (prime number p) of the NTT transform is selected, the value and computation order of twiddle factors are also fixed. In our hardware design, We compute the fixed twiddle factors by software and store it in ROM table as Fig. 1(e). In order to reduce the fan-out and improve the routing congestion, the twiddle factor ROM is also constructed in parallel with multiple smaller ROMs. The data width of a single ROM is 52-bit and depth 64. Twiddle factors are stored directly in ROM in a fixed read order. Thus, twiddle factors can be read sequentially to BFU input ports, as shown in Fig. 1(e). The distribution logic of factors is simple and consumes fewer Mux resources.

4) Pipelined Architecture of Parallelism NTT: As Algorithm 2 shows, in each round of NTT transform, the polynomial coefficients to be calculated are independent and there is no data dependence between BFUs. This property determines that we can do pipelining to further accelerate the polynomial multiplication.

In the process of NTT transform with parallelism 8, each clock cycle requires 16 polynomial coefficients. In Type 2 computations as Fig. 1(c), 16 coefficients can be read from parallel RAMs by same address in one cycle. However, the 16 polynomial coefficients required for BFU are stored in different addresses in Type 1, the two inputs of a single BFU need two clock cycles to be read out. Such property of Type 1 will introduce bubbles in the pipeline, affecting the full utilization of BFUs and the computation delay. In our design, we introduce the ping-pong register structure. By alternating updating registers, our pipeline design can achieve full BFU utilization every cycle without pipeline bubbles.

The registers value in Type 1 pipeline structure are updated every two clock cycles. In Fig. 2, the numbers "0, 16,..." represent the address of data in RAM, the symbols "-,+" represent the identification of the front or back half of 16 coefficients. The gray part in picture is the register configuration and the "D-unit" means the fixed distribution unit we used. As Fig. 2, the coefficients read from RAM are stored in register A in odd clock cycles. In even clock cycles, the coefficients are stored in register B. We fetch the coefficients from read ports of RAM and register A at even clock cycles. Both the RAM and register A are all parallel stored 16 coefficients within one line, we fetch half of coefficients from them (8 coefficients) respectively. The fetched 16 coefficients will be input to the

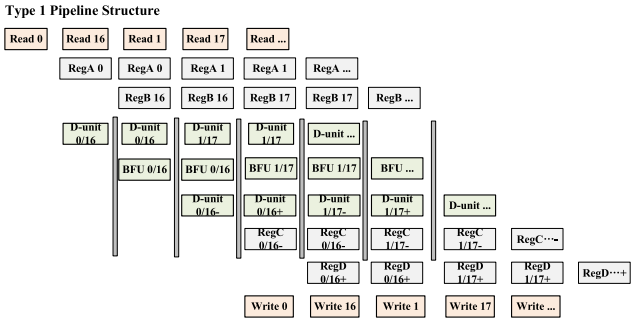

Fig. 2. The pipeline structure of Type 1-NTT and register configuration that improves the utilization of BFUs.

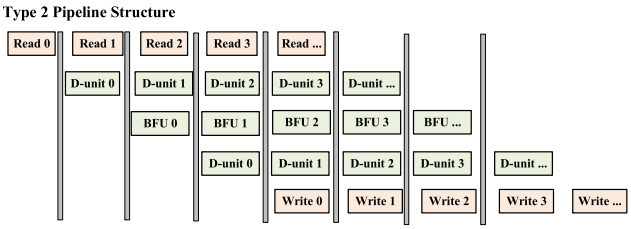

Fig. 3. The pipeline structure of Type 2-NTT.

Distribution unit then participate in the BFU computations. At the odd cycles, we fetch the other half coefficients from register A, B as above. Through such "ping-pong" register structure, the full utilization of BFUs can be achieved after the pipeline is filled. After the BFU computations, the computed coefficients should be written back to the original address (in-place NTT). The register configuration for write-back are similar to read register structure as Fig. 2 shows.

For Type 2 NTT rounds, the 16 coefficients read from RAM are exactly the inputs required by BFUs. The read-out coefficients are directly input to the distribution unit without additional registers as Fig. 3.

Compared with non-pipeline design, the clock cycle consumption of NTT are reduced from $(2 + 2 + 2T_{d-unit} + T_{BFU})\frac{N}{2*8} \cdot \log N$ to $(2 + 2T_{d-unit} + T_{BFU} + \frac{N}{2*8}) \cdot \log N$ by pipeline design in Fig. 2,3. Algorithm 2 shows that the INTT transform is the inverse process of NTT. Thus, the INTT unit can be implemented with NTT design methods, such as the parallelism, fixed distribution, and pipeline. In order to save the computational resource, INTT and NTT units use BFUs together. The outputs of BFUs are selected by transform mode. Fig.1(f) is the general BFU circuit for NTT and INTT.

B. Simplified Design of Modular Reduction

In addition to the algorithm optimization that defines complexity and time cost, the computation within the BFU is also necessary to optimize. The polynomial coefficients are transformed into Z_p in NTT/INTT. Therefore, the computations in NTT/INTT are always executed within Z_p, the modular reduction is required after each computation. As a computational unit adopted after each coefficient multiplication, the optimization of modular reduction can improve the overall performance of the hardware design.

1) Modular Reducion: The modular reduction in NTT/INTT can be divided into two types: one is the modular reduction after addition or subtraction, and the other is the modular reduction after multiplication. Since the result computed by addition/subtraction values at most $2p$, the modular reduction can be calculated by simple judgment circuit. However, the calculation of modulo p after multiplication is the most complex computation in BFU. The calculation of modulo p is $x - p\left\lfloor \frac{x}{p} \right\rfloor$, Where x is the input value and p is a prime. The division in modulo p has high complexity and large overhead that we should avoid. Montgomery algorithm and Barrett reduction are commonly used for modular reduction, the core of this type algorithm is to transform the division into other simpler operations. Compared with Montgomery algorithm, Barrett reduction can calculate approximate result with less cost and lower computational complexity, but it still include the high-cost multiplications. As a computational module frequently called by BFU, modular reduction module has significant resource and clock cycle overhead. The optimization of it can bring considerable benefits to BFU. If the multiplication/division in the reduction algorithms can be avoided, the computation can be greatly speeded up [26].

2) Selected a Solinas Prime: In this design, we propose a simplified modular reduction with lower consumption by selecting a special Solinas moduli to further optimize the polynomial multiplication. The Solinas prime as $p = 2^x \pm 2^y \pm 1$ format, can simplified the modular reduction by special relationship between parts $2^x, 2^y \ldots$ [16]. Our hardware implementation selects the latest suggested security parameter set of TFHE [29]. According to the parameter set, the degree of polynomials $N = 1024$, the coefficient widths of two multiplication polynomials are 32-bit and 7-bit, respectively. This means in order to prevent the computation overflow in NTT/INTT transform, the width of the modulo p must be greater than $32 + 7 + 10 + 1 = 50$ bits, and some computation margin should be considered. Since the negative-wrapped convolution NTT/INTT [15] is selected to optimize the polynomial multiplication, the prime modulus p must have primitive $2Nth$ root of unity where $\omega^{2N} \equiv 1(\mathrm{mod}\, p)$.

(1) Observing the simplified modular reduction algorithm, we find that the core of such algorithms is calculates an estimated value. The Barrett algorithm is a modular reduction algorithm with low complexity. However, it still needs multiplication to calculate the estimated value. Therefore, we consider low Hamming weight prime p for NTT modulo. In this way, due to the properties of special prime, the estimated value can be calculated by simple shift, addition and subtraction without multiplication.

(2) Since the modulo width determines the calculation scale in NTT, the prime (meet all requirements) with a smaller width is a better choice. As shown in Table II, in the exploration of several special number categories, we find a 52-bit Solinas prime p as $p = 2^{52} - 2^{12} + 1 = 4503599627366401$. This prime not only satisfies the width and unity root requirements of NTT, but also has the property of low Hamming weight.

TABLE II
THE COMPARISON OF SPECIAL NUMBERS

Special Number	Width	Prime or not
Mersenne		
M1∼M12	≤39-bit	prime
M13	157-bit	prime
Fermat		
F0∼F4	≤ 17-bit	prime
F5∼F7	≤ 129-bit	not prime
Goldilock		
X1∼X25	≤ 50-bit	/
X26∼X37	≤ 74-bit	not prime
X38	76-bit	prime
Goldilock(+)		
X25∼X31	≤ 62-bit	not prime
X32	64-bit	prime
Solinas		
$2^{52} - 2^{12} + 1$	52-bit	prime

1: Goldilock(+) refers to $p = 2^{2n} - 2^n + 1$
2: $Xn \rightarrow 2^{2n} - 2^n \pm 1$

(3) In this way, we can use the property of $p = 2^{52} - 2^{12} + 1$ to remove the multiplication/division in modular reduction (as Equation (1)-(4)). The modular reduction can be calculated only by shift, addition and subtraction. For highly parallel BFUs, NTTs and polynomial multiplication modules, such computational simplification and resource optimization can improve the efficiency of bootstrapping scheme to a certain extent.

1: Goldilock(+) refers to $p = 2^{2n} - 2^n + 1$
2: $Xn \rightarrow 2^{2n} - 2^n \pm 1$

Let x be the input value (multiplication product 104-bit), moduli $p = 2^{52} - 2^{12} + 1$. Then x can be written as (1). Where a, b, c are 12-bit, 40-bit, 52-bit numbers, respectively.

$$x = a \cdot 2^{92} + b \cdot 2^{52} + c \tag{1}$$

Due to the relationship in (2),(3), the modular reduction of $xmodp$ can be calculated as (4):

$$2^{92} \bmod p = (2^{52} - 2^{40}) \bmod p \tag{2}$$

$$2^{52} \bmod p = (2^{12} - 1) \bmod p \tag{3}$$

$$x \bmod p = -a \cdot 2^{40} + (a + b) \cdot 2^{12} + (c - a - b) \tag{4}$$

In this way, the result of $xmodp$ can be calculated completely by addition, subtraction and shift operation without multiplication. Our modular reduction algorithm is shown in Algorithm 3. Different from [16], our design finds a larger modulo (52-bit), then the recommended single-stage delay of pipelined multiplier is about 52-bit addition latency. Therefore, we do not further split the reduction logics, but utilize the pre-computation & selection structure to obtain modular reduction results. This modular reduction design avoids the expensive multiplication, which can effectively reduce the resource cost and further improve the speed of polynomial multiplication.

In modular reduction module: Firstly, we find a 52-bit Solinas prime p as $p = 2^{52} - 2^{12} + 1$ that meets all the requirements (width for correctness, unity root) mentioned above. Compared with the commonly used Barrett reduction and other bigger primes, this design has less resource consumption,

Algorithm 3 The Optimized Modular Reduction for $p = 2^{52} - 2^{12} + 1$

Input: 104-bit x, moduli p
Output: 52-bit $v = x \ mod \ p$
1: $a \leftarrow x[103:92]$ $b \leftarrow x[91:52]$ $c \leftarrow x[51:0]$
2: $u_0 = -a$ $u_1 = a + b$ $u_2 = c + p$
3: $v_0 = u_0 \ll 40$ $v_1 = u_1 \ll 12$ $v_2 = u_2 - u_1$
4: $v' = v_0 + v_1 + v_2$
5: if $v' > p$:
6: $v' \leftarrow v' - p$
7: else:
8: Return $v \leftarrow v'$

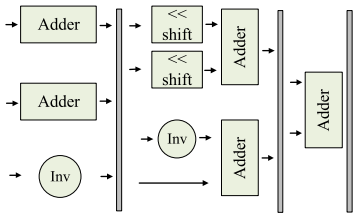

Fig. 4. The pipeline design of Modular Reduction.

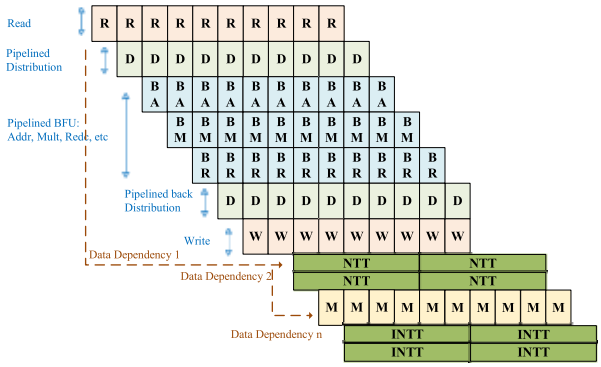

Fig. 5. The High-level Pipelined Polynomial Multiplication.

which is helpful to the scheme acceleration. For highly parallel BFUs, NTTs and polynomial multiplication modules, such computational simplification and resource optimization can improve the efficiency of bootstrapping scheme to a certain extent; Secondly, the modular reduction is decomposed with chosen Solinas prime, and the logic distribution is specifically optimised to achieve superior computational effectiveness with balanced resources; Finally, the pipelined modular reduction with appropriate single-stage delay is designed to match the full-pipelined NTT, further accelerating the computation.

3) Pipeline Modular Reduction: As Fig. 4 shows, we adopt pipeline to implement the modular reduction unit, to match the high-level pipeline structure of whole polynomial multiplication module and further reduce the clock cycle cost. In pipelined modular reduction, the delay of single pipeline stage is about 52-bit addition latency.

In polynomial multiplication module, all of the functional unit are pipelined to reduce the clock cycle cost and speed up the computation. The high-level pipeline structure of polynomial multiplication is shown in Fig. 5. Due to the limitation of picture space, the pipeline inside the functional units are not expanded. In Fig. 5, the "R, W" represent the read and write of RAM, "D" represents the distribution unit which contains a 3-stage pipeline. "BA" is also a pipelined design which composed by logic units such as addition, subtraction and judgment in BFU, it's single pipeline stage delay is also about 52-bit addition. "BM" represents the multiplication in BFU, it uses the multiplier IP core of FPGA. We adopt DSP mode to implement the multiplier with 12-stage pipeline to match the 52-bit addition single stage delay. "M" is the digit-multiplication after NTT transform, which is also implemented by the above multiplier IP core. As Fig. 5, all of the units and their internal computations are pipelined. However, different NTT-rounds introduce data dependency problem, the inputs of next NTT round are the outputs of previous round. Between different data-dependency, we allocate enough memory to ensure the execution of this pipelined polynomial multiplication unit.

Algorithm 4 Key Switching Procedure

Input: LWE cipher $c_{s',m} = (\mathbf{a}', b)$, and a switching key $KS_{s' \to s}$, $s' \in \{0,1\}^{n'}$, $s \in \{0,1\}^{n}$, and precision parameter t
Output: LWE cipher $c_{s,m} = (\mathbf{a}, b)$
1: Let $\overline{a_i}'$ be the closest multiple of $\frac{1}{2^t}$ to a_i', $\left|\overline{a_i}' - a_i'\right| < 2^{-(t+1)}$
2: Binary decompose $\overline{a_i}' = \sum_{j=1}^{t} a_{i,j} \cdot 2^{-j}$, $a_{i,j} \in \{0,1\}$
3: Return $c_{s,m} = (\mathbf{0}, b') - \sum_{i=1}^{n'} \sum_{j=1}^{t} a_{i,j} \cdot KS_{i,j}$

C. The Design of Key Switching Module

In this part, we describe the design of key switching in detail. As Algorithm 1 shows, the input of key switching module is the ciphertext output by "BlindRotate" computation. Key switching can convert the ciphertext c' (encrypted under s') to c (encrypted under s) with the same plaintext m. The key switching algorithm in TFHE is shown in Algorithm 4.

1) On-Chip Switching Key Generation: According to the parameter set we selected, the size of switching key KS is about 300Mb. Due to the parallelism of polynomial multiplication unit, the "Matrix" module in "BlindRotate" step needs to receive and store 12Kb bootstrapping key per clock cycle. The data transmission resource and memory on the chip are already very tight. However, the computational overhead of switching key KS generation is minimal and can be calculated only by addition and subtraction. Thus, generating switching keys on-chip is considered. Generating KS on-chip saves at least half of transmission resources and memory at low computational cost, which improves design frequency and further optimizes overall performance.

We sample initial key vector by software under the uniform distribution, and the noise vector is sampled under the Gaussian distribution. These two random vectors together with the old key, are packed in ROM table and stored on the chip. These

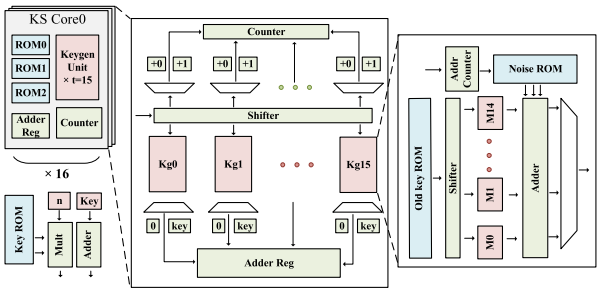

Fig. 6. The Architecture of Key Switching Module with Multiple Parallelism.

vectors are used as the seeds of key generation to compute the switching key.

2) Key Switching Module: The input cipher of key switching module is the "BlindRotate" output. As the two parties for data transfer, the "BlindRotate" and the key switching module require identical bandwidth to avoid the bandwidth conversion. As we detailed above, we set the "BlindRotate" that occupies the chief clock cycle consumption in bootstrapping procedure as the center of optimization. The parallelism of "BlindRotate" and its internal modules are designed depending on platform resources and parallelism experiments. Consequently, to match the sender module "BlindRotate"s bandwidth (data width 32-bit×parallel output 16 wires), we set the parallelism of key switching module to 16.

The architecture of key switching module with multiple parallelism is shown in Fig. 6. The left part of Fig. 6 is the top-level key switching structure, which consists of 16 parallel key switching units. The middle part shows how the 16 parallel units compute the key switching. Based on the fact that the $KS[0-629]$ of all KS keys are equal, we choose to store the switching computation numbers in a counter, and use the truncated multiplier to implement the accumulated subtractions (line 3 in Algorithm 4) in key switching procedure. In this way, a great amount of clock cycles can be saved. As Fig. 6 shows, the 16 key switching units generate their own key (the only different value $KS[630]$) respectively, and then, accumulate the keys or not by the value of $a_{i,j}$ (0 or 1). In this way, the data dependency between units introduced by accumulated subtractions can be eliminated, thus avoiding the frequent data transfer between units and reducing the routing pressure. The right part in picture is the design of key generation unit. The key generation only consumes addition, the computational cost is low, so we consider further parallel computation. It can be seen from Algorithm 4 that every $a_{i,j}$ requires at most t times accumulation. Therefor, we set $t = 15$ parallel key generation blocks in a single key generation unit to further speed up the key switching procedure.

D. Functional Modules Design With Parallelism Matching

In "BlindRotate" step in bootstrapping procedure, the line 5-6 of Algorithm 1 contains "$(X^{-\overline{a_i}} - 1) \cdot ACC$", "$\mathbf{h} \cdot ACC$", "$+ACC$" computations. Reducing the clock cycle consumption of these functional modules can effectively accelerate the bootstrapping. Analyze the data flow in "BlindRotate" step, as described above, the parallelism of

Algorithm 5 $(X^{-\overline{a_j}} - 1) \cdot ACC$ Procedure

Input: Polynomial $ACC \rightarrow (ACC_0, ACC_1, \ldots) \in \mathbb{T}_N[X]$, and a_j from LWE cipher
Output: Product polynomial $P \rightarrow (p_0, p_1, \ldots) \in \mathbb{T}_N[X]$

If $a_j < N$:
  For $i = 0$ to $a_j - 1$:
    $p_i = -ACC_{i-a_j+N} - ACC_i$
  For $i = a_j$ to $N - 1$:
    $p_i = ACC_{i-a_j} - ACC_i$
Else :
  For $i = 0$ to $a_j - N - 1$:
    $p_i = ACC_{i-a_j+2N} - ACC_i$
  For $i = a_j - N$ to $N - 1$:
    $p_i = -ACC_{i-a_j+N} - ACC_i$
Return $P \leftarrow (p_0, p_1, \ldots, p_{N-1})$

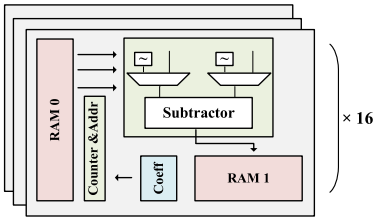

Fig. 7. The design of parallel "Polyminus" module.

polynomial multiplication is set to 8. This means that the polynomial ACC transmits 16 coefficients in one clock cycle during computation. In order to avoid bandwidth transformation and unnecessary resource cost, the parallelism of other functional modules in "BlindRotate" step all match to 16, 16 input coefficients are calculated at the same time.

1) Polyminus Module: For convenience of description, we call "$(X^{-\overline{a_i}} - 1) \cdot ACC$" as "Polyminus" in the following. Analyze the calculation in "Polyminus Module", the property of $(X^{-\overline{a_i}} - 1)$ multiply polynomial makes this polynomial multiplication can be calculated by only selecting coefficients. The computation procedure is shown in Algorithm 5.

As the architecture of Fig. 7 shows, after 16 coefficients are input to module, the counters generate read address of each coefficient according to the coefficient order respectively. Due to the parallelism 16, the 16 copies of input polynomial are stored in RAM0 to meet the computational bandwidth. The switched result is stored in RAM1. RAM0 in "Polyminus Module" is only updated by the input polynomial, and RAM1 is only updated by computation results. Such memory configuration also allow "Polyminus Module" to be pipelined to further accelerate the computation.

2) Overall Architecture: According to the Algorithm 1, the overall bootstrapping can be divided into several modules as Fig. 8. The computations in "Initial Module" (Algorithm 1 line 1-4) and "Extract Module" (Algorithm 1 line 7) just shift operation and read/write of RAM, so these two modules are not detailed in Fig. 8.

The input LWE cipher load into bootstrapping scheme by PCI-Express. After the "Initial Module", the cipher input

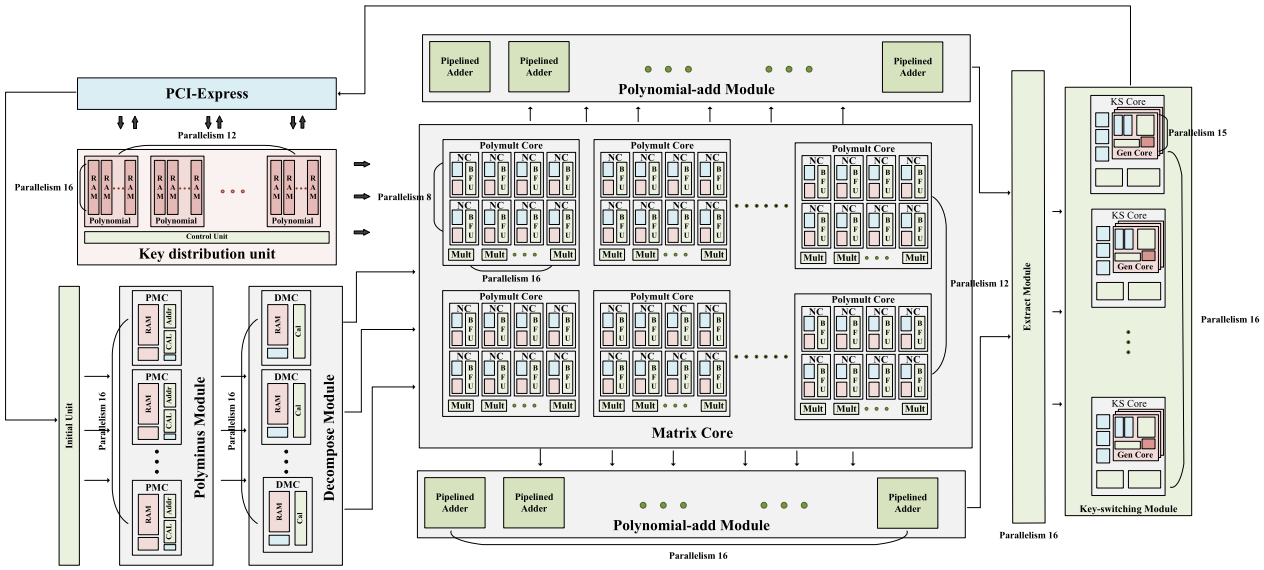

Fig. 8. The overall architecture of bootstrapping scheme with multiple parallelism matching.

to "Polyminus Module" for "BlindRotate" step as Fig. 8. The modules in gray background compose the "BlindRotate" module as Algorithm 1 line 5,6. The "Polyminus Module" and "Decompose module" both set 16 parallelism to match the bandwidth of the polynomial multiplication. Meanwhile, due to the data dependency in module, both "Polyminus Module" and "Decompose module" are designed with pipeline to further accelerate the computations.

The design of "Polynomial multiplication" is detailed in section III-A. It is noteworthy that the BK_i is decomposed to a 6×2 matrix. Each element in matrix is a polynomial with degree 1024. So in this part, we configure 12 polynomial multiplication modules to parallel compute the $BK_i \cdot ACC$ as "Matrix module" in Fig. 8. The two columns of product matrix need to be added up respectively according to $BK_i \cdot ACC$. Two parallel "Polynomial-add module" is set to compute these additions. In order to match the bandwidth of polynomials, there are 16 "add-core" inside the "Polynomial-add module". Since a single add-core needs to calculate the addition of 6 polynomial coefficients, the add-core is pipelined with the single-stage delay of 32-bit addition to further accelerate the procedure.

The results of "BlindRotate" are extracted and input to "Key-switching module". To match the polynomial bandwidth these two modules are also constructed 16 parallel units respectively. The "Key generation unit" inside the "Key switching core" is paralleled with precision parameter $t = 15$ as section III-C described. The cipher after bootstrapping will output through PCI-Express.

Considering the resources and characteristics of the hardware platform we selected, we proposed such a multiple parallelism-matching hardware design with high-level pipeline to optimize the bootstrapping procedure.

(1). Because the polynomial multiplication has the largest clock cycle overhead in bootstrapping, we set NTT as the center of parallelism design. It can bring considerable benefits to global bootstrapping performance.

(2). Some modules can not utilize parallel and pipeline technologies naturally. We conduct various structural design and low-level logic design in each module to achieve parallelism-matching and optimize the bootstrapping performance.

(3). In limited-resources hardware platform, parallelism-matching design leads to heavy hardware burden. After exploration of several experiments, according to the characteristics and overhead of each computation in bootstrapping, we set a appropriate parallelism for NTT and selected several modules (whose gain in latency is greater than the increased overhead) to conduct parallelism-matching. Considering the global bootstrapping performance in parallelism-resource tradoff condition, our design can effectively reduces the clock cycle cost and improves the bootstrapping performance.

E. Analysis of Key Unrolling

As mentioned above, the bottleneck of TFHE bootstrapping performance is the serial "BlindRotate". Key unrolling method [20] utilizes the property of $s_i \in \{0, 1\}$, to calculate more $X^{-\overline{a_i}}$ in a single round to reduce the loops of "BlindRotate". The serial "BlindRotate" can be partially paralleled through this method. However, key unrolling technology accelerate bootstrapping at the cost of expanded memory consumption. The truth table of unrolling factor $m = 2$ as Fig. 9. We can see that the loops n of "BlindRotate" reduced to $n/2$ by $m = 2$ key unrolling, but the cost is about 3 times of memory resource. Key unrolling have the tradeoff between resource and latency.

Our hardware scheme designed with almost globally high parallelism. The high-level pipelines are used to optimize the high clock consumption modules. This makes our resources in hardware platform have been relatively tight. Without key unrolling ($m = 1$), our hardware RAM consumption has reached 65%. Due to the memory requirements of key unrolling, we analyze the memory consumption of our hardware scheme to further accelerate bootstrapping. We tried to

s_{2i}	s_{2i+1}	$X^{a_{2i}s_{2i}+a_{2i+1}s_{2i+1}}$
0	0	1
0	1	$X^{a_{2i+1}s_{2i+1}} = (1 + s_{2i+1}(X^{a_{2i+1}} - 1))$
1	0	$X^{a_{2i}s_{2i}} = (1 + s_{2i}(X^{a_{2i}} - 1))$
1	1	$X^{a_{2i}s_{2i}+a_{2i+1}s_{2i+1}} = X^{a_{2i}+a_{2i+1}}$

Fig. 9. The truth table of unrolling factor $m = 2$.

further exploit the utilized RAM resource on chip to place the unrolling key, saving the on-chip SRAM resources.

As we described above, the polynomials in hardware scheme are stored in 16 parallel small RAMs due to the parallelism-matching. The smallest RAM IP in FPGA is RAMB18E2 which can store 18k-bit data. However, the polynomial coefficient RAM only consumes 3k-bit data. 80% of the utilized RAM storage is empty and wasted. In key unrolling design, we increase the storage of a single utilized RAM (but still in 18k-bit range) to store the key added by key unrolling. In this way, the added key does not require additional instantiation of RAM IP, but is stored in the memory that has been previously instantiated and not used. Therefore, the utilization of RAMs is more effective and the SRAM resource on-chip can be saved.

In addition, the key unrolling introduced more "Polyminus Module" parallel in hardware scheme. We note that the data bandwidth of parallel "Polyminus Module" increases with the unrolling factor. Furthermore, since we store the added key in utilized RAM IP, the depth of a single RAM increases, and the control signal fan-out becomes larger accordingly. Therefore, with the unrolling factor increases, the route congestion gradually appears and affecting the timing closure. When m is selected to 3, the frequency of hardware scheme decreases slightly. Greater performance can be obtained by more routing resources on-chip. The performance of schemes with various unrolling factors is shown in section IV.

IV. Experimental Results

This section mainly describes the implementation and performance of our hardware schemes and gives the comparison with previous designs. We implement FPGA-based bootstrapping schemes with various unrolling factors. Besides FPGA implementation, the core computational module "NTT transform unit" are also implemented individually by TSMC standard cell library in 28nm CMOS for performance comparison.

A. Parameter Set

The implementation we proposed designed with the latest parameter set suggested by TFHE. The LWE/R-LWE security corresponding to this parameter set reaches 128-bit/130-bit respectively, so the overall scheme security achieves 128-bit. The parameter set is as follows (1). The LWE dimension $n = 630$, it also represents the loops number of "BlindRotate". The degree of polynomial $N = 1024$ determines the security of R-LWE. $L = 3$, $Bg = 1024$ determine the decomposition accuracy and the $t = 15$ determines the precision of key switching. They all affect the security of bootstrapping system.

(1)$n = 630$, $N = 1024$, $L = 3$, $Bg = 1024$, $t = 15$

Other parameter sets selected by previous implementations are shown as follows for comparisons.

(2)$n = 500$, $N = 1024$, $L = 2$, $Bg = 1024$, $t = /$;

(3)$n = 592$, $N = 2048$, $L = 3$, $Bg = 128$, $t = /$;

(4)$n = 500$, $N = 1024$, $L = 3$, $Bg = 1024$, $t = /$;

The security comparison of these parameters sets is shown in Table III.

TABLE III
THE SECURITY COMPARISON OF PARAMETER SETS

Parameter set	n	N	L	Bg	t	Security(λ $bits$)
(1)	630	1024	3	1024	15	128
(2)	500	1024	2	1024	/	110
(3)	592	2048	3	128	/	80
(4)	500	1024	3	1024	/	110

TABLE IV
THE PERFORMANCE COMPARISON OF NTT TRANSFORM

Design	[25]	[26]	[24]	Our(FPGA)	Our(ASIC)
q(bits)	28	16	28	52	52
N	1024	1024	1024	1024	1024
Latency(us)	2.00	101.84	0.92	1.61	0.56
Freq.(MHz)	125	183	215	433	800
LUT/ATP	132K/264.0	3K/305.52	94.4K/86.85	33K/53.13	/
FF/ATP	59K/118.0	3K/305.52	104.5K/96.14	44K/70.84	/
DSP/ATP	448/0.896	58/5.907	640/0.589	160/0.257	/
BRAM/ATP	96/0.192	29/2.953	80/0.074	39.5/0.063	/
Area	/	/	/	/	1022528.7794
Power	/	/	/	/	1014.5mW

B. Baselines

We implemented our scheme on Virtex Ultrascale+ VCU128 FPGA. Besides FPGA implementation, the core computational module "NTT transform unit" are also implemented individually by TSMC standard cell library in 28nm CMOS for performance comparison. We compared our design with state-of-art implementations include CPU, GPU, ASIC and FPGA designs. The CPU design [29] based on Intel Core i7-4910MQ at 2.90 GHz. The NVIDIA GeForce RTX 3090 GPU @ 1.70 GHz is used as GPU design [28] baseline. The $ASIC$ design based on 16nm PTM process [12]. The baseline of FPGA design [11] for comparison is Xilinx Virtex UltraScale+ VU13P FPGA.

C. Performance

The core computational module "NTT transform" is implemented on both FPGA platform and ASIC. The NTT performance and the comparison are shown in Table IV. The hardware design of "NTT transform" runs at 433MHz on Xilinx Virtex UltraScale+ VCU128 FPGA. The latency of a complete NTT transform is 1.6us.

It can be seen from the table that, under the same polynomial degree, the coefficient in our design have almost twice width compared with other designs. This is determined by bootstrapping parameters as section III-B described. The design of [24] with 32 parallelism of NTT has the largest DSP consumption. Considering the platform resources and the

matching-parallelism in scheme, we only construct 8 parallel BFUs in our design. Greater performance can be obtained by selecting a larger hardware platform. The NTT transform unit we proposed have at least 30% improvement in ATP with 52-bit coefficient width. Synthesized by Synopsys Design Compiler with TSMC 28nm CMOS cell library, our NTT unit can achieve the frequency of 800MHz, the latency of a complete NTT transform is only $0.56us$.

In TFHE bootstrapping procedure, on one hand, the clock cycles consumed by polynomial multiplication account a large part of the entire overhead, which nearly determines the speed of bootstrapping. Therefore, accelerating polynomial multiplication and NTT as much as possible can bring considerable benefits to the bootstrapping scheme. On the other hand, bootstrapping hardware system is a large-scale design with high computational complexity and large key storage, the crossed data wires of multi-BFUs in parallelized NTT modules further increases the complexity of the bootstrapping system. So how to design an NTT module that is more lightweight is also an important issue to be addressed.

For our NTT module in bootstrapping: (1) We designed parallelized NTT modules with multi-BFU inside, parallel computing can effectively reduce the clock cycle overhead; (2) We proposed fixed distribution access pattern NTT to optimize the routing complexity and improve the frequency of hardware design; (3) The high-level pipelines are utilized in NTT to match the polynomial multiplication pipeline and further speed up the bootstrapping.

In this way, our design can execute NTT transform and polynomial multiplication more efficiently as shown in Table IV. Faster NTT helps to speed up the bootstrapping, and the lower resource cost of lightweight NTT permits higher parallelism in hardware design. Thus, our NTT design can effectively improves the bootstrapping performance.

We proposed implementations of bootstrapping on FPGA platform with key unrolling factor $m = 1, 2, 3$ respectively. The performance and FPGA resource utilization are shown in Table VI. The design with $m = 1, 2$ running at 180 MHz. Due to the increased fan-out and bandwidth introduced by key unrolling, the scheme with unrolling factor $m = 3$ running at 170MHz.

As described in section III-E, by storing the added key in previously wasted memory, we can implement key unrolling without new RAM IP utilization as table shows. However, on the one hand this kind of cross-module data storage makes the control logic more complex, leads to more crossover data wires. Due to fan-out problem and routing congestion, too large unrolling factor is not beneficial to timing closure. On the other hand, when we select a larger unrolling factor, it will introduce more key. If the introduced key consumption greater than previously wasted space, the new RAM utilization is still required. At this point, key unrolling is still constrained by platform resources. Therefore, blindly unrolling the bootstrapping key cannot result in a stable latency reduction. The implementations we proposed with unrolling factor $m = 1, 2, 3$ can compute a complete bootstrapping within $6.62ms, 3.31ms, 2.35ms$, respectively.

D. Compared With CPU and GPU Designs

The CPU design in Table VI is proposed by TFHE library [29]. On Intel Core i7-4910MQ at 2.90 GHz, the CPU design obtained a running time of $13ms$ per bootstrapping [9], this work is referred to as "TFHE library" in Table VI. Compared to this CPU design, the $75\%, 82\%$ latency reduction can be achieved by our design with rolling factor $m = 2, 3$. Without key unrolling as this CPU design, the latency reduction of our implementation can achieve up to 49%. Another CPU baseline is proposed in [14] and is referred to as "Concrete" in Table VI. With the higher security, our design have $9.36\times - 34.17\times$ speed up compared to Concrete CPU baselines.

The cuFHE (reference tag same "cuFHE") [28] proposed a bootstrapping scheme without key unrolling. On NVIDIA GeForce RTX 3090 GPU @ 1.70 GHz [11], the bootstrapping latency of this GPU design is $9.34ms$. With the higher security, our design can achieve $29\% - 75\%$ speed improvement compared with cuFHE.

E. Compared With ASIC Design

The only previous ASIC design of MATCHA (reference tag same "MATCHA") on $16nm$ PTM process needs more than $6.8ms$ to complete a NAND bootstrapping. Our design achieves $2.5\% - 65\%$ improvement with higher security. It worth noting that, the GPU version of MATCHA benefits from the high-memory-bandwidth up to $640GB/s$ and the multiple parallel aggressive BKU. Due to the substantial memory access in bootstrapping computation, if the higher bandwidth can be obtained, our design would achieve better performance. Furthermore, the platform resources limits the parallelism we designed. Faster bootstrapping can be obtained in our design by higher parallelism with more resource on-chip.

F. Compared With FPGA Designs

There are only two previous FPGA implementations of complete TFHE bootstrapping. One of which is [11], the [11] proposed bootstrapping with parameter set (2)(3) on Xilinx Virtex UltraScale+ VU13P FPGA. We refer to this work by "YKP22" as author initials in Table V,VI. Our design can achieve performance improvement up to 65% with higher security and larger computation scale (The prime modulu in NTT is selected only 32-bit in YKP22). It worth noting that, our design select 52-bit NTT prime modulus, the data width is almost twice compared with YKP22 in NTT module. In addition, due to the tight BRAM resource, we did not utilize RAM IP to implement all the FIFOs in design, but instead build them by FF. It results in high FF overhead and low utilization of RAM resources. If more BRAM resources are used, the consumption of FF can be reduced.

The other FPGA baseline is SPSL21 [13]. This implementation does not utilize neither pipeline nor other parallel structure. It takes extremely long time to compute bootstrapping. Accordingly, the resource consumed by SPSL21 design is low. Compared with SPSL21, our implementation have at least $37\times$ ATP reduction with $2k\times$ speed up.

TABLE V
THE BANDWIDTH COMPARISON OF VARIOUS DESIGNS

	This paper	YKP[11]	Concrete[14]	FPT[31]	cuFHE[28]	SPSL[21]	MATCHA[12]
Platform	Xilinx VCU128	Xilinx VU13P	AMD Ryzen 3990X	Xilinx Alveo U280	GeForce RTX 3090	Xilinx 7Z020	16nm PTM
Peak External Memory Bandwidth	31.5GB/s (PCI-e4.0 16 lanes)	77GB/s (DDR4)	200GB/s (DDR4)	112GB/s (HBM2)	1008GB/s (GDDR6X)	77GB/s (DDR4)	640GB/s (HBM2)
Bandwidth factor	1	2.44	6.35	3.56	32	2.44	20.32

TABLE VI
THE COMPARISON OF BOOTSTRAPPING LATENCY AND CONSUMPTION

Design	Security λ $bits$	Clock (MHz)	Latency (ms)	Latency Reduction	m	LUT/ATP	LUT ATP Reduction	FF/ATP	FF ATP Reduction	DSP/ATP	DSP ATP Reduction	RAM/ATP	RAM ATP Reduction
TFHE library[29]	128	/	13.00	49%	1	/	/	/	/	/	/	/	/
Concrete[14]	110	/	62.00	9.36×	1	/	/	/	/	/	/	/	/
Concrete[14]	80	/	80.30	34.17×	1	/	/	/	/	/	/	/	/
cuFHE[28]	110	/	9.34	29%	1	/	/	/	/	/	/	/	/
MATCHA[12]	110	/	> 6.80	2.5%	1	/	/	/	/	/	/	/	/
SPSL21[13]	110	/	17640	2.6k×	1	36K/635.04	200×	24K/423.36	90×	40/705.60	37×	2.9Mb/51.16	160×
YKP22[11]	110	180	7.53	12%	1	925K/6.96	−54%	729K/5.49	−14%	6240/47.0	−59%	319Mb/2.40	−87%
YKP22[11]	110	180	3.76	12%	2	842K/3.17	−56%	662K/2.49	+6%	7202/27.1	−65%	338Mb/1.27	−87%
YKP22[11]	110	180	2.51	6%	3	569K/1.43	−10%	448K/1.12	+75%	6640/16.7	−59%	383Mb/0.96	−83%
YKP22[11]	80	180	19.13	65%	1	931K/17.81	−82%	728K/13.92	−66%	6272/119.98	−84%	343Mb/6.56	−95%
YKP22[11]	80	180	9.56	65%	2	770K/7.36	−77%	602K/5.76	−54%	6446/61.62	−85%	400Mb/3.82	−96%
YKP22[11]	80	180	6.38	62%	3	534K/3.41	−62%	419K/2.67	−26%	6034/38.50	−82%	429Mb/2.74	−94%
FPT[31]*	128	200	0.48	/	/	526K	/	916K	/	5494	/	17.5Mb	/
FPT[31]*	110	200	0.58	/	/	595K	/	1024K	/	5980	/	14.5Mb	/
Proposed1	128	180	6.62	/	1	480K/3.17	/	715K/4.73	/	2881/19.07	/	48Mb/0.32	/
Proposed2	128	180	3.31	/	2	510K/1.69	/	801K/2.65	/	2881/9.54	/	48Mb/0.16	/
Proposed3	128	170	2.35	/	3	544K/1.28	/	843K/1.98	/	2881/6.77	/	67Mb/0.16	/
Proposed4	110	200	2.13	/	2	414K/0.88	/	625K/1.33	/	1281/2.73	/	40Mb/0.09	/
Proposed5	110	200	1.43	/	3	510K/0.73	/	750K/1.07	/	1281/2.73	/	56Mb/0.08	/

FPT [31] is a FPGA implementation for streaming CMUX kernel ("BlindRotate") with Chisel [32] based on Xilinx Alveo U280 datacenter accelerator. Like SystemVerilog, Chisel (an open-source HDL embedded in Scala) is a full-fledged HDL with direct constructs to describe synthesizable combinational and sequential logic. FPT [31] optimized the computational scale of bootstrapping by fixed-point FFT (with extra noise), so that the deep pipeline structure can be implemented on FPGA. The core computational module FFT is constructed bySGen which employs concepts introduced in Spiral FFT IP Core generator [33]. We refer to this work by "FPT [31]" in Table V,VI. The "*" means that FPT is a incomplete scheme, it only implemented CMUX ("BlindRotate") in bootstrapping without key switching. We implemented complete bootstrapping scheme "Proposed 4,5" under the same parameter set as FPT.

We use PCI-e as the interface of data transfer in all of our designs. We instantiate the Xilinx ip core XDMA to build the PCI-express path. In bootstrapping scheme with small unrolling factors (m=1,2), we use PCI-e 3.0 protocol with 16 channels for key transfer, the peak bandwidth of this interface is 15.74GB/s. Due to the larger transfer in scheme with larger unrolling factor (m=3), we adopt PCI-e 4.0 protocol with peak bandwidth 31.5GB/s. The peak bandwidth comparison is shown in Table V.

Note that the peak bandwidth of our scheme is only 31.5GB/s. While, the FPT use HBM2 Pseudo Channels(PCs) to transfer keys. With 8 channels, the FPT achieved bandwidth of 112GB/s. It can be seen that, the bandwidth of other designs larger than ours by at least 2.4 times. If higher bandwidth are available, our design would achieve better performance by faster data-transfer. Set our peak transfer bandwidth 31.5GB/s as benchmark "1", and the bandwidth factors of other designs are 2.44/6.35/3.56/32/20.32, respectively. We calculate the bootstrapping latency times bandwidth factors as a metric, under this, our hardware scheme can achieve at least 30.5% faster than other designs includes FPT. The main overhead of our data transfer is PCI-e resources.

The main optimizations in FPT are all based on the computation scale reduction by fixed-point FFT. The fixed-point FFT introduces extra noise in CMUX computation. In designs utilized FFT as Concrete [14] and MATCHA [12], noise monitoring or experimental results are provided to illustrate the decryption correctness. However, FPT only has the assumption of $\sigma_{tot}^2 = \sigma_{FFT}^2 + \sigma_{IFFT}^2 + \sigma_{BK}^2$ to match the theoretical noise bounds with variable-controlling measurement, it does not provide the experimental noise value introduced in CMUX and the failure probability of decryption. Our hardware design utilize precise NTT with integer arithmetic to accelerate polynomial multiplication. In bootstrapping procedure, NTT does not introduce extra noise. In our scheme, the noise variance set completely same with [29] by TFHE proposer to guarantee the decryption correctness.

FPT has certain requirements for hardware platforms about bandwidth and memory. However, our design focused on low-level optimization, bootstrapping schemes with different platforms or parameter sets can be optimized by our methods. The design we proposed is more flexible in application.

FPT does not implemented complete bootstrapping due to the leak of key switching. In bootstrapping algorithm, the scale of key switching key KS is at least same as the bootstrapping key BK. Our design implements complete bootstrapping with key switching. The on-chip key generation module we designed can reduce the data transfer scale and improve the bootstrapping efficiency. If a larger hardware platform is selected, we can store more bootstrapping keys on

chip. The more parallel computation lanes can be built and the higher pipeline levels can be set, which will greatly improve design efficiency and accelerate our bootstrapping.

V. Conclusion

In this paper, we proposed the first FPGA implementation of TFHE bootstrapping with 128-bit security to our acknowledgement. We designed the fixed distribution access pattern NTT with high-level pipeline to accelerate the polynomial multiplication. The simplified modular reduction has been proposed to optimize the computation. Our implementation with multiple parallelism-matching architecture can further speed up the bootstrapping procedure. The hardware bootstrapping scheme we implemented can achieve $5.5\times - 34\times$ speedup compared with the state-of-art CPU baselines. The experimental results indicate that, compared with previous FPGA design, our design achieves a latency improvement of $6\% - 65\%$ with higher security.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, May 2009, pp. 169–178, doi: 10.1145/1536414.1536440.

[2] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. 3rd Innov. Theor. Comput. Sci. Conf.*, Jan. 2012, pp. 309–325, doi: 10.1145/2090236.2090262.

[3] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, 2012, Art. no. 144. [Online]. Available: http://eprint.iacr.org/2012/144

[4] D. Dharani and M. Rohit, "Research on homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Intell. Syst. Commun., IoT Secur. (ICISCoIS)*, Feb. 2023, pp. 409–437, doi: 10.1109/iciscois56541.2023.10100464.

[5] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. CRYPTO*, vol. 8042. Heidelberg, Germany: Springer, 2013, pp. 75–92, doi: 10.1007/978-3-642-40041-4_5.

[6] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Proc. 34th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, Apr. 2015, pp. 617–640, doi: 10.1007/978-3-662-46800-5_24.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Proc. ASIACRYPT*, vol. 10031. Heidelberg, Germany: Springer, 2016, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-662-53887-6_1

[8] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE," in *Proc. ASIACRYPT*, vol. 10624. Cham, Switzerland: Springer, 2017, pp. 377–408. [Online]. Available: https://doi.org/10.1007/978-3-319-70694-8_14

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.

[10] I. Chillotti, D. Ligier, J.-B. Orfla, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE," in *Proc. ASIACRYPT*, vol. 13092. Cham, Switzerland: Springer, 2021, pp. 670–699. [Online]. Available: https://doi.org/10.1007/978-3-030-92078-4_232021

[11] T. Ye, R. Kannan, and V. K. Prasanna, "FPGA acceleration of fully homomorphic encryption over the torus," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2022, pp. 1–7, doi: 10.1109/HPEC55821.2022.9926381.

[12] L. Jiang, Q. Lou, and N. Joshi, "MATCHA: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 235–240, doi: 10.1145/3489517.3530435.

[13] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, "An FPGA-based programmable vector engine for fast fully homomorphic encryption over the torus," in *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*, 2021. [Online]. Available: https://par.nsf.gov/biblio/10282639

[14] I. Chillotti, M. Joye, J. D. Ligier, J.-B. Orfla, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending TFHE," in *Proc. 8th Workshop Encrypted Comput. Appl. Homomorphic Cryptogr. (WAHC)*, vol. 15, 2020.

[15] F. Winkler, *Polynomial Algorithms in Computer Algebra* (Texts & Monographs in Symbolic Computation). Cham, Switzerland: Springer, 1996, doi: 10.1007/978-3-7091-6571-3.

[16] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT," *IACR Trans.* , vol. 2020, no. 2, pp. 49–72, 2020., doi: 10.13154/tches.v2020.i2.49-72.

[17] J. M. Pollard, "The fast Fourier transform in a finite field," *Math. Comput.*, vol. 25, no. 114, pp. 365–374, 1971.

[18] D. Knuth, *The Art of Computer Programming 2: Seminumerical Algorithms*, MA, USA: Addison-Wesley, 1968.

[19] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.

[20] T. Zhou, X. Yang, L. Liu, W. Zhang, and N. Li, "Faster bootstrapping with multiple addends," *IEEE Access*, vol. 6, pp. 49868–49876, 2018.

[21] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020, doi: 10.1109/TC.2020.2988765.

[22] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Washington, DC, USA, Feb. 2019, pp. 387–398, doi: 10.1109/HPCA.2019.00052.

[23] D. D. Chen et al., "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 62, no. 1, pp. 157–166, Jan. 2015, doi: 10.1109/TCSI.2014.2350431.

[24] T. Ye, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "FPGA acceleration of number theoretic transform," in *High Performance Computing*. Springer, 2021, pp. 98–117.

[25] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, M. Becchi, and A. Aysu, "A flexible and scalable NTT hardware: Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 346–351, doi: 10.23919/DATE48585.2020.9116470.

[26] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt, "Exploring energy efficient quantum-resistant signal processing using array processors," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Barcelona, Spain, May 2020, pp. 1539–1543, doi: 10.1109/ICASSP40776.2020.9053653.

[27] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 1295–1309, doi: 10.1145/3373376.3378523.

[28] (2019). *CUDA-Accelerated Fully Homomorphic Encryption Library*. [Online]. Available: https://github.com/vernamlab/cuFHE

[29] (2020). *TFHE: Fast Fully Homomorphic Encryption Library Over the Torus*. [Online]. Available: https://github.com/tfhe/tfhe

[30] M. V. Beirendonck, J.-P. D'Anvers, and I. Verbauwhede, "FPT: A fixed-point accelerator for torus fully homomorphic encryption," Cryptol. ePrint Arch., Paper 2022/1635, 2022. [Online]. Available: https://eprint.iacr.org/2022/1635

[31] J. Bachrach et al., "Chisel: Constructing hardware in a scala embedded language," in *Proc. DAC Design Autom. Conf.*, Jun. 2012, pp. 1212–1221, doi: 10.1145/2228360.2228584.

[32] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 17, no. 2, pp. 1–33, Apr. 2012, doi: 10.1145/2159542.2159547.

[33] Xilinx. (2022). *UG1302—VCU128 Evaluation Board User Guide (UG1302) (V1.2)*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1302-vcu128-eval-bd