Acceleration of the Bootstrapping in TFHE by FPGA

Jian Zhang, Aijiao Cui, *Senior Member*, *IEEE*, Yier Jin, *Senior Member, IEEE*

Abstract— Privacy-preserving computing is playing an ever-increasingly important role in various fields. A leading example of privacy-preserving computing is Fully Homomorphic Encryption (FHE). FHE enables arbitrary computations directly on the ciphertext. This guarantees that the original data will not be disclosed while processing the data. However, FHE brings in the high computation cost which, in turn, limits the application of FHE. Among all steps of FHE, bootstrapping is a critical operation yet a bottleneck for the FHE efficiency. Torus FHE (TFHE) was presented as a method which can compute arbitrary Boolean functions on ciphertext with fast gate bootstrapping. In this paper, we show an implementation of TFHE gate bootstrapping on ZYNQ ZCU102 FPGA board. The memory operation is specially organized to facilitate the implementation of the adopted Number Theoretic Transform (NTT) of external product. Each function involved in the TFHE gate bootstrapping is implemented at the register-transfer level (RTL), and each operation is carefully scheduled to maximize the parallelism. Experimental results show that with ZCU102 working at the frequency of 300MHz, the proposed scheme can bootstrap one bit within 1.9ms on average. Compared with the accelerated TFHE using the mainstream CPU, the proposed scheme shows a 5.0X speedup. If under the similar clock frequency, it presents 1.23X faster than cuFHE which is accelerated by GPU. The proposed scheme also shows other advantages such as high efficiency and better tradeoff than existing FPGA-based acceleration schemes.

Index Terms—Fully homomorphic encryption, bootstrapping, NTT, FPGA

1 Introduction

Nowadays, data resources have become indispensable to human society. The emergence of big data technology has caused a revolution around the world. The circulation of data has always played a vital role in mining the value of data. However, it also causes great concerns for data security. Encryption algorithms can be adopted to safeguard data during communication from an eavesdropping third-party. However, traditional encryption algorithms are insufficient to protect sensitive data hosted by third parties, such as in the cloud. Privacy-preserving computing [1] secures data that is intended to be available, but also invisible to malicious third-party hosts. It can guarantee data security both during communication and in processing, thereby assuring complete data privacy. Privacy-preserving computing has been adopted in multiple areas to secure data. For example, in the financial industry, institutions can leverage privacy-preserving computing to evaluate the risk that one customer poses to another if they were to break a contract. In the medical field, privacy-preserving computing enables multiple institutions to analyze the medical cases without releasing patient data. Furthermore, the government can rely on privacy-preserving computing to supervise and secure data to facilitate deep mining in an effort to improve community welfare.

Many techniques have thus been developed to implement privacy-preserving computing, includingMulti-party Secure Computing (MPC) [2], Trusted Execution Environments (TEEs) [3], and Fully Homomorphic Encryption (FHE) [4]. FHE refers to encryption systems that have fully homomorphic properties. FHE leverages a public-key encryption scheme to guarantee data security during communication, while its fully homomorphic property enables the encrypted data to be directly computed upon. With FHE, the owner of the data resource can encrypt the sensitive data and transmit it to the untrusted party, who can directly compute upon the encrypted data (ciphertext) and then return to the owner the computed ciphertext. The owner can then decrypt the returned data to obtain the computation result he or she desires. Thus, FHE achieves ideal privacy-preserving computation. However, it is non-trivial to implement FHE. How to implement FHE in an efficient way therefor becomes a great concern.

In 1977, Rivest et al. first proposed the concept of homomorphic encryption (HE) [5]. Partial Homomorphic Encryption (PHE) schemes, which can only implement homomorphism on the arithmetic of addition or multiplication, were later proposed as RSA [6] and Paillier [7] encryption schemes, respectively. In 2009, Gentry proposed the first fully homomorphic encryption scheme [8]. In this scheme, a SomeWhat Homomorphic Encryption (SWHE) scheme was first built. SWHE can implement homomorphism on addition and multiplication for several rounds. Then, the operation of bootstrapping is used to refresh the ciphertext so that infinite rounds of homomorphic operations can be performed on the ciphertext. For FHE, Bootstrapping is indispensable for the construction of FHE, while it also turns to the most complex and expensive operation in FHE.

Some SWHE schemes have been proposed to overcome the low computational efficiency of FHE, such as BGV [9],

- J. Zhang and A. Cui are with Harbin Institute of Technology (Shenzhen), Shenzhen, China. (E-mail: zhangjian2023@foxmail.com; cuiaj@hit.edu.cn)
- Y. Jin is with University of Science and Technology of China, Hefei, China. E-mail: jinyier@ustc.edu.cn
- J. Zhang and A. Cui contribute equally to this article. Corresponding author is A. Cui.

FV [10] and CKKS [11]. These schemes uniformly improved the computational depth of SWHE to meet the need of actual computing so that the non-trivial bootstrapping in FHE could be avoided. These SWHE schemes can implement certain rounds of homomorphic addition and multiplication on ciphertext at a high speed. However, they have low flexibility and the setting of parameters are very complicated. These weaknesses make SWHE schemes sufficiently inefficient for complex computing scenarios, such as deep learning.

Torus FHE (TFHE) [12] is a well-known FHE scheme. The plaintext in TFHE just consists of the Boolean values '1' and '0'. Also, only Boolean operations are involved in TFHE homomorphic computation. Compared with SWHE, which can only implement homomorphic addition and multiplication, TFHE shows eminent flexibility. TFHE can also implement bootstrapping efficiently. These properties make TFHE an ideal FHE scheme which can implement any computation in an infinite depth. The homomorphic computation in TFHE is implemented via gate bootstrapping.

TFHE is also known as CGGI16 [12] and the source code of TFHE library is published in [13]. The published TFHE library can bootstrap 1 bit in 52ms. In 2017, the original TFHE scheme was improved and named as CGGI17 [14]. The time for gate bootstrapping was reduced to 13ms. Later, Zhou [15] further shrunk the time on bootstrapping to 10ms by optimizing multiple addends. In 2018, Dai [16] disclosed the source of cuFHE, which implemented the CGGI17 scheme on graphic processing units (GPUs). Number Theoretic Transform (NTT) was adopted to accelerate the multiplication between two polynomial rings. Bootstrapping of the scheme was optimized with its overhead reduced to 0.5ms [16]. Despite GPUs having abundant computing resource, they are still general-purpose instruction-based processors. The implementation of a simple instruction involves a series of operations. Thus, the performance of gate bootstrapping based on GPUs has little room for improvement. In comparison, an accelerator designed for TFHE gate bootstrapping may provide better opportunity to improve the efficiency as each primary operation in bootstrapping can be scheduled and implemented more efficiently. Such an accelerator will undoubtedly facilitate the application of TFHE in industries [17].

The work in [18], denoted by HSCD, explored the co-design of software (SW) and hardware (HW) to implement the gate bootstrapping. The scheme only implements the most complicated computation using a Field Programmable Gate Array (FPGA), while all other operations are completed by software. Thus, a global optimization cannot be achieved among all operations and the operations performed by software have little room for optimization. The work in SPSL [19] proposed a customizable vector engine, which can be augmented with application-specific custom instructions. However, the optimization of the polynomial multiplication, which is the most time-consuming step in bootstrapping is ignored. Also, the parallel computing is not adopted. The scheme YPK in [20] proposed to decompose TFHE gate bootstrapping into several parameterized primitives to enable more parallelism and optimization. YPK also includes the bootstrapping key unrolling scheme which can be used to achieve tradeoffs between the performance and the consumed resource to meet different constraints. The NOMP in [21] denotes a heterogeneous scheme which uses both a CPU and FPGA. Multiple *Tgate* processing units, called TGCs, are introduced to improve the parallelism. A CPU is used since the scheduling of these TGCs is not trivial. The FPT scheme in [22] proposes a fixed-point accelerator on a streaming processor to implement the TFHE bootstrapping. FPT uses noise-trimmed fixed-point representations which are up to 50% smaller than the implementations using floating-point or integer FFTs. The FPT's microarchitecture is built as a streaming processor that facilitates the cascading of operation stages and the bootstrapping pipeline. MATCHA [23] denotes a TFHE accelerator implemented in ASIC. It implements a pipelined datapath consisting of TGSW clusters and external product cores to enable aggressive bootstrapping key unrolling that invokes FFTs and IFFTs less frequently during a bootstrapping operation.

We note that these FPGA-based implementation in [18]-[20] rely on high level synthesis (HLS). HLS tools help developers use high-level description to synthesize circuits within short developing time. However, the performance of the circuits synthesized through HLS is usually less optimized than those synthesized directly using the register-transfer level (RTL) code. Further, circuits generated through HLS have deeper pipelines, leading to high resource consumption. RTL can provide more potential for the global optimization and it can benefit the implementation on hardware much more than HLS, as indicated in [22]. However, a global RTL optimization is not fully explored in existing FPGA-based implementations [18]-[22].

In this work, we therefore propose a scheme to implement an FPGA accelerated gate bootstrapping in TFHE using RTL. RTL optimization on each operation involved in bootstrapping is fully explored. Operations are carefully scheduled to maximize the parallelism and facilitate the pipeline among multiple modules. The NTT [24] and Montgomery modular multiplication [25], [26] are adopted to accelerate the complex multiplication of polynomial rings and multiplication of Galois numbers, respectively. The memory is specially organized to facilitate the most complicated operation of NTT. The contributions in this work can be summarized as follows:

1. We develop an RTL design of the TFHE bootstrapping acceleration scheme. We customize each bootstrapping primitive in addition to the most time-consuming NTT operation and fully consider the parallelism and cooperation among its phases. The BRAM is also carefully organized at RTL to facilitate the primary operation of NTT. These deep customizations at RTL benefit the global optimization of the acceleration scheme, hence the performance. The theoretical time for bootstrapping of one ciphertext isreduced to 1.9ms, which is more efficient than most acceleration schemes on both FPGAs and general-purpose processors.

2. We implement the entire bootstrapping scheme on an FPGA using DMA for key transmission. The ZYNQ UltraScale+MPSoC ZCU102 platform is used to simulate gate bootstrapping in TFHE. This not only reduces the cost of

software and host processor resources, but also allows for cooperation between software and hardware. As such, our implementation achieves better tradeoff between performance and resource consumption. Additionally, developers can follow this scheme to implement an efficient bootstrapping on a semi-custom FPGA or full-custom ASIC.

The rest of this paper is organized as follows. Section II introduces the preliminaries and the algorithm of bootstrapping in TFHE. Section III presents the hardware implementation of bootstrapping. Section IV evaluates the performance of the proposed scheme and compares it with other schemes on different hardware processors. Finally, this paper is concluded.

2 Preliminaries on Bootstrapping in TFHE

A. The role of bootstrapping in TFHE

In an FHE scheme, as shown in Fig. 1, a customer cannot process his data D using the function f. He has to ask an untrusted party to perform f on D while he desires to keep his private data D invisible to the untrusted party. To do so, he then encrypts D with his secret key K to generate a ciphertext CT and passes it with the function f to the party for the computations. The party will evaluate f on CT to obtain a computed ciphertext CT'. The evaluation of f here indicates an implementation of f with homomorphism so that the customer can decrypt CT' to obtain the computed data. As CT' equals the encrypted $f(D)$ under K, the customer can finally decrypt CT' to obtain the desired result $f(D)$. It can be seen that the homomorphic evaluation of the function f plays a key role in the overall FHE process. Bootstrapping is hence developed and adopted in FHE to implement the homomorphic evaluation.

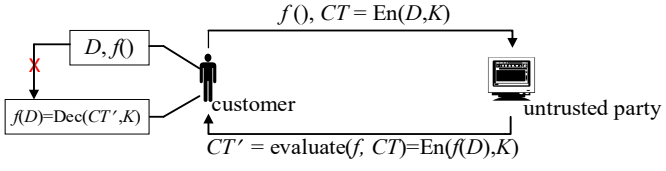
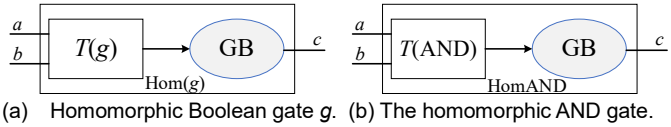


Fig. 1. The principle of FHE.

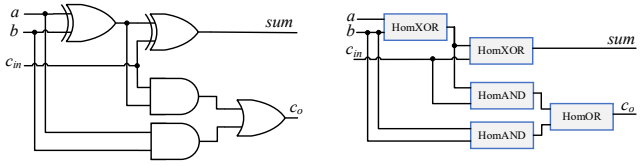
In TFHE, the message space is defined as {0, 1/2}, which amounts to two Boolean values of TRUE and FALSE. The evaluated function f in whatever depth is first expressed in a Boolean logic. Then, the Gate Bootstrapping (GB) operation is adopted in each Boolean logic gate to implement a homomorphic gate. As shown in Fig. 2(a), to implement a logic gate g with homomorphism, its inputs are first transformed as specified by $T(g)$ and the output of $T(g)$ is then processed by GB. For example, a homomorphic AND gate can be implemented by the combination of T(AND) and GB, as shown in Fig. 2(b).



(a) Homomorphic Boolean gate g. (b) The homomorphic AND gate.

Fig. 2. GB in a homomorphic gate.

With each homomorphic Boolean gate, the evaluation of f can be achieved by replacing each logic gate in f with its corresponding homomorphic gate. Thus, the overall Boolean function has homomorphism. For example, for the homomorphic evaluation of a full adder, as shown in Fig. 3(a), five logic gates in the adder are substituted by two HomXORs, two HomANDs and one HomOR gates. The bootstrapped full adder can then implement the homomorphic addition between two inputs and the carrier, which are all in the form of ciphertexts.



(a) A full adder design. (b) The homomorphic full adder.

Fig. 3 The gate bootstrapping scheme for a full adder.

Precisely speaking, the gate bootstrapping in TFHE consists of the operations of bootstrapping and Public-Key Switching [27]. The former costs more than 90% of the computing resources while the cost of the latter is negligible. Our proposed work solely focuses on the bootstrapping in GB. For clarity, the terms bootstrapping or gate bootstrapping are used alternatively in this article to refer to the bootstrapping in GB.

B. The implementation of bootstrapping in TFHE

Multiple ciphertexts are involved in bootstrapping. To help better understand the process of bootstrapping, the ciphertexts computed by three public-key cryptosystems with homomorphism are first introduced. Then, some key operations in bootstrapping are elaborated prior to the presentation of the overall algorithm.

1) The ciphertexts in TFHE

Every public-key cryptosystem relies on some problems that are believed to be computationally difficult, such as the integer factorization problem and the discrete logarithm problem. These two problems can be solved in polynomial time on a quantum computer. As a secure alternative, the Learning With Error (LWE) problem was introduced by Regev in 2005 [28] and become a fundamental problem in lattice-based cryptography. FHE refers to a series of public-key cryptosystem which have homomorphism. Although the encryption schemes involved in the mainstream FHE systems are slightly different, they are uniformly built on the problem of LWE [28].

Three different LWE-based cryptosystems, Torus LWE (TLWE), Torus RingLWE (TRLWE) and Torus RingGSW (TRGSW), are involved in the TFHE bootstrapping to generate three types of ciphertexts, also called samples. As shown in Fig. 4, the original message M is first encrypted by TLWE under the input secret key $\boldsymbol{tK}$ to obtain a TLWE sample $\boldsymbol{tc}$. A TRLWE sample ACC is created to act as an intermediate ciphertext during the bootstrapping process and TRGSW is applied on $\boldsymbol{tK}$ to generate a TRGSW sample as the bootstrapping key $\boldsymbol{\mathcal{BK}}$. Three samples $\boldsymbol{tc}$, $\boldsymbol{\mathcal{BK}}$ and ACC are then used to implement the bootstrapping.

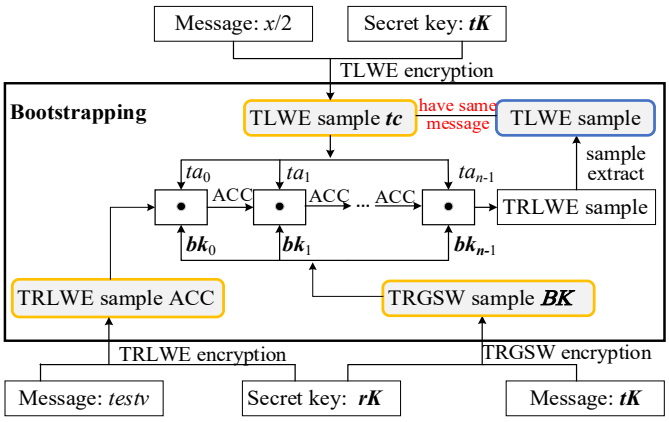


Fig. 4. Three ciphertexts (samples) in bootstrapping.

These ciphertexts in different forms will be briefly introduced below. Note that the detailed encryption algorithms can be found in [9], [28], [29]. The related notations are listed in Table I. A variant in bold fonts denotes a vector. The characters '*t*', '*r*' and '*g*' denote the parameters in TLWE, TRLWE and TRGSW, respectively.

Table I. The Notations in Different Ciphertexts.

Notation	Description
λ	security parameter
$\mathbb{Z}$	The set of integers
$\mathbb{R}$	The set of real numbers
$\mathbb{B}$	The binary set {0,1}
$\mathbb{T}$	The real Torus $\mathbb{R}/\mathbb{Z}$, i.e., the set of real numbers as $\mathbb{R}$ mod 1
$\mathbb{T}_N[X]$	(N-1)-degree Polynomial in $\mathbb{R}[X]/(X^N+1)$ mod 1
$\mathbb{B}_N[X]$	(N-1)-degree Polynomial in $\mathbb{Z}[X]/(X^N+1)$ with coefficients in $\mathbb{B}$

In TLWE encryption, the message space $M = \{0, 1/2\} \subset \mathbb{T}$. A message is selected as $tu \in M$. The secret key is generated as $\boldsymbol{tK} \in \mathbb{B}^n$. The public key can be generated as $(\boldsymbol{ta}, tb) \in \mathbb{T}^{n+1}$, where $\boldsymbol{ta} \in \mathbb{T}^n$ denotes a randomly generated vector. $tb = <\boldsymbol{ta}, \boldsymbol{tK}> + e$, where e denotes an error which is sampled from a gaussian distribution and the symbol $< >$ denotes the inner product between two vectors. The encryption is implemented by $tb = tb + tu$ and the ciphertext is obtained as $\boldsymbol{tc} = (\boldsymbol{ta}, tb) \in \mathbb{T}^{n+1}$, as shown in eqn. (1).

$$\boldsymbol{tc} = \underbrace{(ta_0, ta_1, ta_2, ..., ta_{n-1}, tb)}_{(n+1)\ elements\ in\ \mathbb{T}} \tag{1}$$

TRLWE denotes the general form of TLWE. In TRLWE, a message ru is selected from $\mathbb{T}_N[X]$. The secret key is generated as $\boldsymbol{rK} \in \mathbb{B}_N[X]^k$. The public key is generated as $(\boldsymbol{ra}, rb) \in \mathbb{T}_N[X]^{k+1}$, where $\boldsymbol{ra} \in \mathbb{T}_N[X]^k$ denotes a randomly generated polynomial vector and $rb = <\boldsymbol{ra}, \boldsymbol{rK}> + e$. The encryption is implemented by $rb = rb + ru$ and the ciphertext is obtained as $\boldsymbol{rc} = (\boldsymbol{ra}, rb) = \mathbb{T}_N[X]^{k+1}$. The TRLWE ciphertext $\boldsymbol{rc}$ is a (k+1)-dimension polynomial vector with all its (k+1) elements in $\mathbb{T}_N[X]$, as shown in eqn. (2).

$$\boldsymbol{rc} = \underbrace{(ra_0, ra_1, ra_2, ..., ra_{k-1}, rb)}_{(k+1)\ elements\ in\ \mathbb{T}_N[X]} \tag{2}$$

In TRGSW encryption, the message ru is selected from $\mathbb{B}$. The secret key resembles that in TRLWE as $\boldsymbol{rK} \in \mathbb{B}_N[X]^k$. The public key is formed by (k+1)l TRLWE samples that encrypt zero, as $\boldsymbol{gc} = (\boldsymbol{gc}_0, \boldsymbol{gc}_1, \boldsymbol{gc}_2, \ldots, \boldsymbol{gc}_{(k+1)l-1}) \in \mathcal{M}_{(k+1)l,k+1}(\mathbb{T}_N[X])$, where $\mathcal{M}_{(k+1)l,k+1}(\mathbb{T}_N[X])$ denotes a matrix with (k+1)l rows and (k+1) columns and each element is in $\mathbb{T}_N[X]$. The encryption is implemented by $\boldsymbol{gc} = \boldsymbol{gc} + ru \cdot H$, where H denotes a (k+1)l×(k+1) matrix as shown in eqn. (3) and B_g denotes an positive integer. The TRGSW ciphertext $\boldsymbol{gc}$ is shown in eqn. (4).

$$H = \underbrace{\begin{pmatrix} 1/B_g & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 1/B_g^{\,l} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1/B_g \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1/B_g^{\,l} \end{pmatrix}}_{(k+1)l \times (k+1)\ elements\ in\ \mathbb{T}_N[X]} \tag{3}$$

$$\boldsymbol{gc} = \underbrace{\begin{pmatrix} ga_{0,0}, ga_{0,1}, \cdots, \cdots, \cdots, \cdots, \cdots, ga_{0,k-1}, gb_0 \\ ga_{1,0}, ga_{1,1}, \cdots, \cdots, \cdots, \cdots, \cdots, ga_{1,k-1}, gb_1 \\ \vdots \\ ga_{(k+1)l-1,0}, ga_{(k+1)l-1,1}, \cdots, ga_{(k+1)l-1,k-1}, gb_{(k+1)l-1} \end{pmatrix}}_{(k+1)l \times (k+1)\ elements\ in\ \mathbb{T}_N[X]} \tag{4}$$

2) *Bootstrapping key*

The bootstrapping key is generated by performing TRGSW encryption on the n-element TLWE secret key $\boldsymbol{tK}$. Then, n TRGSW samples are obtained to form the bootstrapping key, $\boldsymbol{BK} = (\boldsymbol{bk}_0, \boldsymbol{bk}_1, \ldots, \boldsymbol{bk}_{n-1})$.

3) *CMUX gate*

In bootstrapping, a design called CMUX gate is introduced to perform a fundamental homomorphic computing, so-called *external product*, which is denoted as $\boxdot$. To obtain an *external product*, a TRLWE sample d and a TRGSW sample C act as two inputs and the output is also a TRLWE sample, as shown in (5). The *External Product* satisfies homomorphism on multiplication.

$$\text{TRLWE} \boxdot \text{TRGSW} \rightarrow \text{TRLWE} \tag{5}$$

The input d contains one row with (k+1) elements while the input C denotes a matrix with (k+1)l rows and (k+1) columns. To implement the multiplication between d and C, the number of columns of d should be enlarged as (k+1)l. Thus, the coefficients of each polynomial in d should be decomposed into l coefficients, as shown in eqn. (6). H and B_g are defined already in eqn. (3).

$$d \boxdot \text{C} = Dec_{H,\,B_g}(d) \boxdot \text{C} \tag{6}$$

Based on *external product*, CMUX gate functions as a homomorphic selector, as shown in eqn. (7).

$$\text{CMUX: C} \boxdot (d_1 - d_0) + d_0 \tag{7}$$

A TRGSW sample C and two TRLWE samples d_1 and d_0 act as three inputs to CMUX. If C is obtained by using TRGSW to encrypt the message '0', CMUX outputs a TRLWE sample whose message equals d_0. Otherwise, the message '1' is encrypted by TRGSW and the CMUX outputs a TRLWE sample whose message is that of d_1.

4) *Bootstrapping algorithm*

The goal of bootstrapping is to obtain a TLWE sample which has the same message as that of the input TLWE sample. It is noted that the TLWE encryption is implemented by $tb=<\boldsymbol{ta},\boldsymbol{tK}>+e+tu$ with the ciphertext obtained as $\boldsymbol{tc}=(\boldsymbol{ta},tb)$. A computation on $\boldsymbol{tc}$, $\varphi(\boldsymbol{tc})=tb-<\boldsymbol{ta},\boldsymbol{tK}>$ can recover the message of $\boldsymbol{tc}$, tu. Bootstrapping is hence designed to implement the computation of $\varphi(\boldsymbol{tc})$. This computation is similar to accumulating $\boldsymbol{ta}$ according to the secret key $\boldsymbol{tK}$. The CMUX gate is adopted as a homomorphic selector to implement such accumulation under the control of a bootstrapping key. Finally, a refreshed TLWE sample is output with its message as that of the input TLWE sample.

The bootstrapping algorithm [27] is shown below. A TLWE sample $\boldsymbol{tc}$, a bootstrapping key $\mathcal{BK}$ and a constant $u_1\in\mathbb{T}$ act as inputs. The bootstrapping key is obtained by TRLWE encryption on the message of TLWE key $\boldsymbol{tK}$ under the secret key $\boldsymbol{rK}=(rk_0, rk_1,\ldots,rk_{n-1})$ [27]. A refreshed TLWE sample $\boldsymbol{rc}$, which has the same message as the input TLWE sample is the final output.

Algorithm 1: Bootstrapping
Input: A constant $u_1\in\mathbb{T}$, $u=u_1/2$;
Input: A TLWE sample $\boldsymbol{tc}=(\boldsymbol{ta},tb)$ with message as $x/2$, $x\in\mathbb{B}$;
Input: Bootstrapping key $\mathcal{BK}$;
Output: A TLWE sample $\boldsymbol{rc}$ with message as $x\cdot u_1$.
Initialize a TRLWE sample ACC=$(\mathbf{0}, testv)\in\mathbb{T}_N[X]^{k+1}$ and $testv=u\cdot(1+X+X^2+\ldots+X^{N-1})\cdot X^{N/2}\in\mathbb{T}_N[X]$;
set $\beta=\lfloor 2N\cdot tb\rceil$, ACC$\leftarrow X^{-\beta}\cdot$ACC;
for j from 0 to n-1 **do**
$\alpha_j=\lfloor 2N\cdot ta_j\rceil$;
C$\leftarrow \boldsymbol{bk}_j$, $d_0\leftarrow$ACC, $d_1\leftarrow X^{\alpha_j}\cdot$ACC
ACC$\leftarrow$CMUX(C,d_1,d_0);
end for
return $(0, u)$ + SampleExtract(ACC)

The algorithm contains two operations as the initialization of a TRLWE sample ACC and a for loop to accumulate ACC under the control of $\mathcal{BK}$. A TRLWE sample ACC is first generated with its message as $testv\in\mathbb{T}_N[X]$. The message $testv$ is obtained by the multiplication between $X^{N/2}$ and an (N-1)-degree polynomial whose coefficients are uniformed as $u=u_1/2$. Note that the multiplication of a polynomial in $\mathbb{T}_N[X]$ with X^i, $i\in Z$, is called rotation of the polynomial by i.

After the initialization, ACC is accumulated. The accumulation is implemented by the rotation of ACC. First, the variant tb in $\boldsymbol{tc}$ is rescaled from $\mathbb{T}$ to $[0, 2N]$ to generate $\beta=\lfloor 2N\cdot tb\rceil$. ACC is rotated by $-\beta$. Then, in each round of the for loop, each variant ta_j in $\boldsymbol{ta}$ is similarly rescaled from $\mathbb{T}$ to $[0, 2N]$ to obtain $\alpha_j=\lfloor 2N\cdot ta_j\rceil$ and ACC is further rotated by α_j. Under the control of the bootstrapping key $\boldsymbol{bk}_i$, CMUX gate determines whether to output the rotated ACC or not. After the accumulation, the first item of the refreshed ACC can be extracted as the output.

The bootstrapping implements the rotation of ACC by $[\,(-\beta+\sum_{j=0}^{n-1}\alpha_j\cdot rk_j+N/2) \bmod 2N\,]$. The rotation operation satisfies homomorphism on TRLWE sample. This indicates that if the TRLWE sample ACC is rotated by i, its message $testv$ is also rotated by i.

It was found in [27] the rotation of ACC can be computed by $I\lceil\varphi(\boldsymbol{tc})\rceil=(\lfloor -2N\varphi(\boldsymbol{tc})\rceil+N/2) \bmod 2N$. We use $u_1=1/2$ as an example to illustrate this equivalence. As $u_1=1/2$, $u=1/4$. The distribution of the message space of the TLWE sample {0, 1/2} can be shown as two dots on $\mathbb{T}$ in Fig. 5. $\varphi(\boldsymbol{tc})$ locates between 0 and 1. If $\varphi(\boldsymbol{tc})\in[0,1/4]\cup[3/4,1]$, $I\lceil\varphi(\boldsymbol{tc})\rceil\in[0, N]$ and the result can be determined as False. It was found that the first item of the refreshed ACC, ACC[0]= $-u$. The minus symbol '-' is thus consistent with the result False. Otherwise, $\varphi(\boldsymbol{tc})\in[1/4, 3/4]$ and $I\lceil\varphi(\underline{\boldsymbol{c}})\rceil\in[N, 2N]$. ACC[0]= $+u$ and the plus symbol '+' agrees with the result True. Therefore, a dotted line passing between the dots 1/4 and 3/4 in Fig. 5 can be used to determine the result of $\varphi(\boldsymbol{tc})$ between True and False. The polarity of ACC[0] coincides with the result. When u_1 changes, the dotted line will vary accordingly.

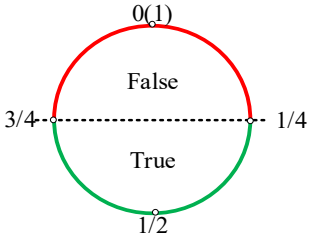


Fig. 5. Distribution of message space {0, 1/2} on $\mathbb{T}$.

5) *Setup of parameters*

Similar to [16], we rescale Torus $\mathbb{T}$ as a 32-bit integer in the range of $[-2^{31}, 2^{31}-1]$. Thus, all operations, except the polynomial multiplication, are performed on the 32-bit integers. The operation of mod 2^{32} is also adopted in the addition to constrain the result within $[-2^{31}, 2^{31}-1]$. The binary set $\mathbb{B}$ is then mapped as $\{-1/8*2^{32},1/8*2^{32}\}$. In TWLE, n = 500, which indicates that 500 rounds of CMUX operations are involved in the bootstrapping. In TRLWE, N=1024 and k=1. In TRGSW, l=2, B_g=1024 and $B_g^2=1024^2$. H is rescaled by factor of 4 and is shown in (8), where $B_1=2^{12}$ and $B_2=2^{22}$.

$$H=\begin{pmatrix}1/B_1 & 0\\ 1/B_2 & 0\\ 0 & 1/B_1\\ 0 & 1/B_2\end{pmatrix} \tag{8}$$

3 The Proposed Implementation of Bootstrapping on FPGA

3.1 *Overview of the architecture*

We propose to implement the bootstrapping in TFHE on the semi-custom design of FPGA so as to maximize the efficiency of bootstrapping. The ZYNQ board is used as the hardware platform. This board mainly includes an ARM processor, DDR SRAM, Direct Memory Access (DMA) and the FPGA ZCU102. As shown in Fig. 6, to mimic the real scenario, we configure the ARM core to simulate a server

which coordinates the overall operation. The FPGA on the ZYNQ board is used as the hardware accelerator for gate bootstrapping. The control signals and data signals between the ARM core and FPGA, shown by the green arrows, are transmitted through the AXI-Lite interface. The DDR is used to store the 15.6Kb ciphertext and the 250Mb bootstrapping key. Under the control of the DDR controller, the data stored in DDR are fetched and transmitted by the central DMA on the FPGA through the AXI-full interface to the BRAM memory and processing unit for bootstrapping. These data paths are shown by the red arrows in Fig. 6.

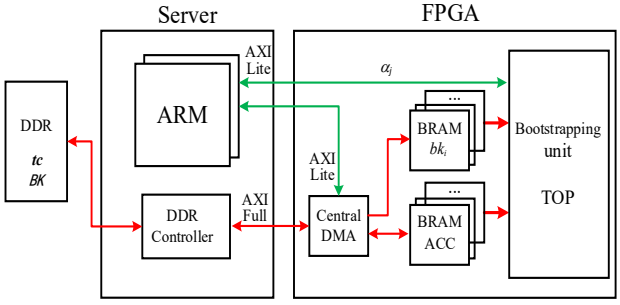


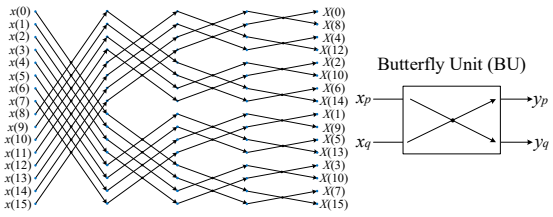
Fig. 6. Architecture of the hardware on the ZYNQ board.

In what follows, we will explore how to customize each bootstrapping primitive, such as rotation, subtraction, decomposition, transformation of integer to Montgomery form, external product and addition with the goal to improve the parallelism and cooperation among phases. The BRAM is also subtly organized at RTL to facilitate the primary operation of NTT at the beginning of bootstrapping.

3.2 *Data organization*

During the bootstrapping, the ciphertext data in the form of TLWE sample will be iteratively accessed and processed. The organization of the data in memory plays an important role in facilitating data accessing and processing. Obviously, organization of data according to the most "expensive" operation can help improve the overall efficiency. It is noted that in bootstrapping, the external product in CMUX, which involves multiplication between polynomials, is the most time-consuming and NTT can reduce the $O(N^2)$ complexity of the schoolbook polynomial multiplication to the quasi-linear complexity of $O(N\log N)$ [34]. NTT is hence adopted and we propose to organize the polynomial coefficients in a manner which can further expedite the NTT operation.

A simple N=16 points NTT example is used to show how to organize the data in block RAMs (BRAMs). As shown in Fig. 7(a), there are 16 coefficients to be processed by NTT in $\log_2 N$=4 stages. Each stage involves $N/2$=8 butterfly operations and each operation is implemented by a Butterfly Unit (BU), as shown in Fig. 7(b).



(a) 4-stage NTT. (b) BU for butterfly operation.

Fig. 7. NTT on N=16 points.

To facilitate the access of data in BRAMs, two inputs to a BU can be stored in the memory units with equivalent offset addresses in two BRAMs. Thus, one BU requires the data from two BRAMs. As all butterfly operations in one stage are independent of each other, multiple BUs can be adopted to operate in parallel to facilitate NTT in one stage. However, the consumed BRAMs and other processing hardware are both multiplied.

Assume that D denotes the depth of a BRAM in an FPGA and P BRAMs are used to store the coefficients. As each polynomial contains N coefficients, the vacancy rate of BRAM can be denoted by $(1\text{-}N/DP)$. Here, D=1024, N=1024. If two BUs are introduced, then four BRAMs are required. This indicates three quarters of BRAMs stay in idle state. If four BUs and eight BRAMs are used for NTT, the efficiency can be 1X higher than that of four BRAMs. However, 7/8 BRAMs are left unused. Also, the hardware for processing coefficients will be doubled. High efficiency, however, aggravates the requirement for the speed of bootstrapping during a key transfer. To achieve a better tradeoff between the efficiency and overhead under the applied FPGA platform, two BUs are introduced as BU_0 and BU_1. If more memory and hardware are available, the parallelism of the proposed scheme can be increased further, along with the efficiency.

Then, 4 BRAMs, as $BRAM_0$ to $BRAM_3$, are applied to store the N coefficients of one polynomial as shown in Fig. 8 to be accessed by two BUs for the NTT operation in Stage 1. The organization of the coefficients in other stages can be deduced in a similar way.

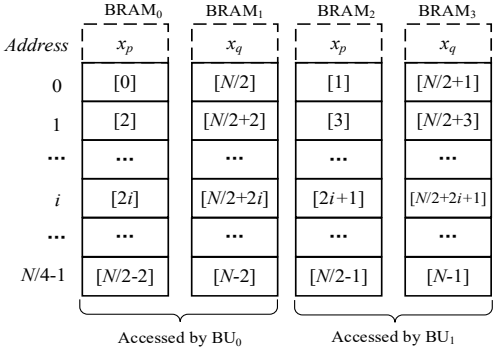


Fig. 8. Initial organization of four BRAMs accessed by two BUs.

As mentioned in Section II, N=1024. As shown in Fig. 9, the 1024 coefficients of a polynomial can be organized with the $2i^{th}$, $(2i+512)^{th}$, $(2i+1)^{th}$, $(2i+513)^{th}$ coefficients stored in $BRAM_0$ to $BRAM_3$, respectively. Here, i denotes the offset address of each memory unit and $i \in [0, 255]$.

The width of each memory unit in BRAM is set as 64-bit, as each 32-bit coefficient needs to be converted to a 64-bit Galois number prior to the NTT operation. Four BRAMs can form an RAM set, called RS1024. Four RS1024s as $RS1024_0$ to $RS1024_3$ are required to implement the overall NTT operation as four polynomials are involved.

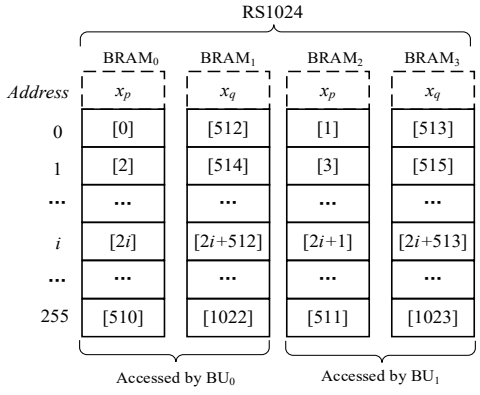


Fig. 9. Organization of 1024 64-bit wide polynomial coefficients in RS1024.

3.3 *Hardware implementation of CMUX*

As shown in Algorithm 1, the bootstrapping algorithm, in essence, consists of applying the CMUX operation n times. The hardware implementation of a CMUX hence determines the efficiency of the overall bootstrapping. The implementation of CMUX can be divided into four phases below according to eqns. (6) and (7). Each operation phase should be designed to guarantee the parallelism and facilitate the cooperation with other phases. The read/write strategy for BRAM data is most important and determines whether the specific parallelism can be implemented successfully.

Phase 1: Rotation and subtraction to compute $d = d_1 - d_0$.

Phase 2: Decomposition of $Dec_{H,Bg}(d)$ and transformation of the 32-bit integers to those in Montgomery form.

Phase 3: Computation of external product.

Phase 4: Computing of Addition.

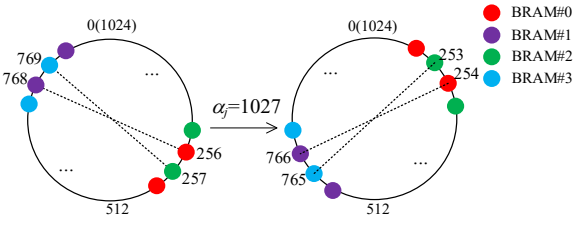
As follows, the implementation in each phase is elaborated.

Phase 1: **Rotation and subtraction to compute $d = d_1 - d_0$.**

As shown in Algorithm 1, d_0=ACC and $d_1 = X^{\alpha_j} \cdot$ACC. As a TRLWE sample, d_0 contains (k+1)=2 polynomials in $\mathbb{T}$ with their coefficients stored in $RS1024_0$ and $RS1024_1$, respectively. d_1 is obtained by rotating the coefficients of each polynomial in d_0 by α_j. The subtraction between d_1 and d_0 can then be performed. If the operations of rotation and subtraction are performed serially, the complexity is $O(N)$ as each polynomial in d_0/d_1 has N coefficients. With N coefficients of one polynomial assigned in four BRAMs, we explore how to process 4 coefficients in parallel in order to reduce the complexity of Phase 1 to $O(N/4)$.

In the proposed scheme, 1024 coefficients of a polynomial in d_0 are stored in four BRAMs of a RS1024. As shown in Fig. 9, the address indices ($2i$, $2i$+512, $2i$+1, $2i$+513) of four coefficients have a uniform offset address i and they can form a group G_i. Then, 1024 indices form 256 groups with $i \in [0, 255]$. When the coefficients indexed by G_i are rotated by α_j, the resulting coefficients can be simply obtained (see [27]) by accessing those coefficients indexed by [($2i$-α_j) mod 1024], [($2i$+512-α_j) mod 1024], [($2i$+1-α_j) mod 1024] and [($2i$+513-α_j) mod 1024], respectively. The mod computation can be implemented on a round "clock", which contains 1024 dots denoting the indices 0 to 1023. As shown in Fig. 10(a), the red, purple, green and blue dots are used to denote the indices of coefficients in $BRAM_0$ to $BRAM_3$, respectively. The "clock" can then be rotated anticlockwise by (α_j mod 1024) to obtain the dots denoting indices of four rotated coefficients. It is not hard to prove that the obtained four dots have different colors, which shows four rotated coefficients also distribute respectively in four BRAMs.

For example, suppose $\alpha_j = 1027$ and i=128. G_{128}= (256, 768, 257, 769) and they can be represented by four color dots connected with two dotted lines in the clock of Fig. 10(a). After the clock is rotated anticlockwise by 3, the four dots in Fig. 10(a) accordingly rotate to new positions as (253, 765, 254, 766), as shown in Fig. 10(b), which indicates that the rotated coefficients can then be accessed from these memories. The rotated four dots are also in four different BRAMs, where the order of four BRAMs varies as $BRAM_2$, $BRAM_3$, $BRAM_0$, $BRAM_1$.



(a) (b)

(a) The dots denoting indices of original coefficients. (b) The dots after the clock is rotated.

Fig. 10. The dots on clock denoting the indices of original and rotated coefficients.

Thus, the rotation of four coefficients in d_0 can be implemented in parallel to compute the offset address of memory storing four coefficients of d_1. The efficiency is improved by three quarters while only requiring a few hardware resources for rotation. The hardware design for the rotation and subtraction in Phase 1 can hence be implemented as shown in Fig. 11.

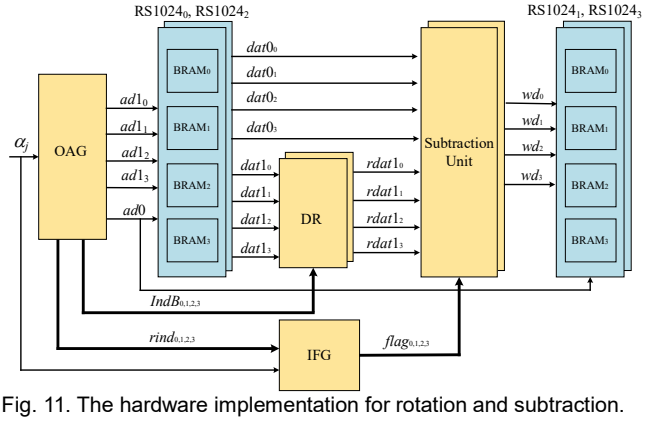


Fig. 11. The hardware implementation for rotation and subtraction.

The hardware design mainly contains four RS1024 units and four control modules as offset address generator (OAG), data reorder (DR), inversion flag generator (IFG) and subtraction. RS1024$_0$ and RS1024$_2$ are used to store the coefficients of two polynomials in d_0 while RS1024$_1$ and RS1024$_3$ store the subtraction result in Phase 1. OAG primarily generates the offset addresses according to the input rotation and the offset addresses of coefficients of d_0. Then, both the coefficients of d_0 and d_1 can be obtained by accessing the memory in RS1024$_0$ and RS1024$_2$ according to their respective offset addresses. After rotation, the order of BRAMs storing the coefficients of d_1 may vary to be different from that of d_0. DR is hence introduced to regulate the BRAMs for d_1 to be in the same order as those for d_0. This not only facilitates the consequent operation of subtraction, but also makes the hardware design economical. The IFG module is used to generate a flag to denote whether the coefficients of d_1 need to be inverted after they are fetched from the BRAMs storing d_0. The subtraction between the four coefficients of d_1 and d_0 is implemented by the subtraction module. Finally, the result is stored into the memory of RS1024$_1$ and RS1024$_3$.

A. Implementation of OAG

The hardware implementation of OAG is shown in Fig. 12. An 8-bit counter is used to generate the offset address i for the coefficients of d_0. Then i is doubled with one bit shifted left. The doubled result is added with 0, 512, 1 and 513, respectively and then subtracted by α_j to generate the exact addresses of four coefficients of d_1, from $rind_0$ to $rind_3$. Each exact address contains 10 bits. In a BRAM assignment unit (BAU), as shown in Fig. 13(a), the 10 bits are divided into two parts. The least significant bit (LSB) and the most significant bit (MSB) unite to determine the index of BRAM, $IndB_0$ to $IndB_3$ for the coefficients in d_1 while the other eight bits indicate their offset addresses adr_0 to adr_3 in BRAM. Each pair is then input to an address order unit, AOU, as shown in Fig. 13(b). Guided by the index of each BRAM, AOU can output the corresponding offset address $ad1_i$ ($i \in [0..3]$) so that the four coefficients of d_1 can be accessed from four BRAMs accurately.

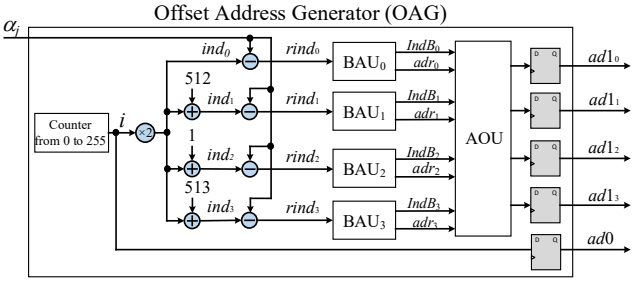
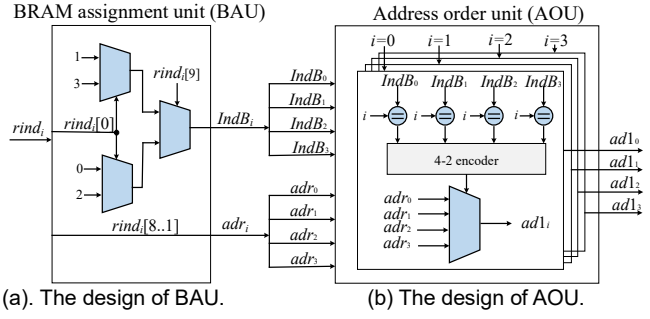


Fig. 12. The design of Offset Address Generator (OAG).



(a). The design of BAU. (b) The design of AOU.

Fig. 13. The design to generate the address in each BRAM for the coefficients of d_1.

B. Implementation of DR

DR unit is introduced to regulate the coefficient data of d_1 from each BRAM to match that of d_0. Its implementation is shown in Fig. 14. It consists of four multiplexers. The data output from four BRAMs are all input to each multiplexer. Under the control of each $IndB_i$, the four coefficients of d_1 are selected to be output so that they can be united with each coefficient of d_0 for the following subtraction.

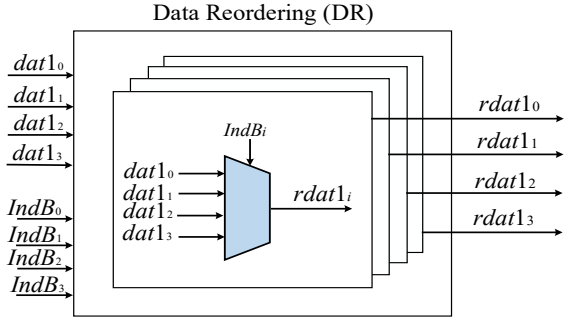


Fig. 14. The design of Data Reorder (DR) unit.

C. Implementation of IFG

During the generation of the coefficients of d_1, if $\alpha_i \neq 0$, the coefficients of d_0 need not only be rotated but also inverted. When α_i is smaller than N, those coefficients with indices less than r ($r = \alpha_i \bmod N$) should be reversed. Otherwise, $\alpha_i > N$ and those coefficients with indices not smaller than r should be reversed. As α_i contains 11 bits, its 10th bit indicates whether it is greater than N=1024. r can be directly computed as $\alpha_i[9:0]$. Thus, we use comparators and XOR gates to implement the inverse flag generator (IFG) unit to generate the negation flags, as shown in Fig. 15.

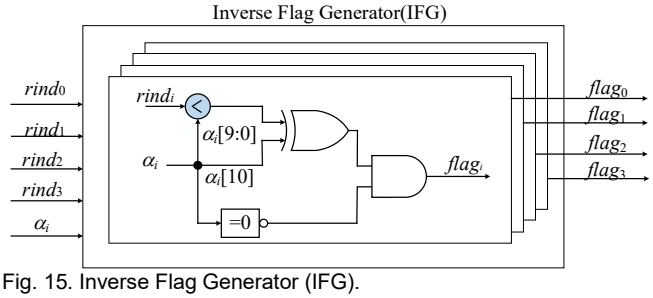


Fig. 15. Inverse Flag Generator (IFG).

D. The implementation of subtraction module

The subtraction module includes four subtractors. Each subtractor is composed of an adder, an inverter and a multiplexer, as shown in Fig. 16. One coefficient of d_0, $dat0_i$ and its corresponding coefficient of d_1, $rdat1_i$ are both input to one subtractor. Under the control of the inverse flag from IFG, the coefficient of d_1 is reversed or maintained. Then, two coefficients are subtracted and the result is stored in the memory in $RS1024_1$/$RS1024_3$ with the offset address as $ad0$ output from OAG.

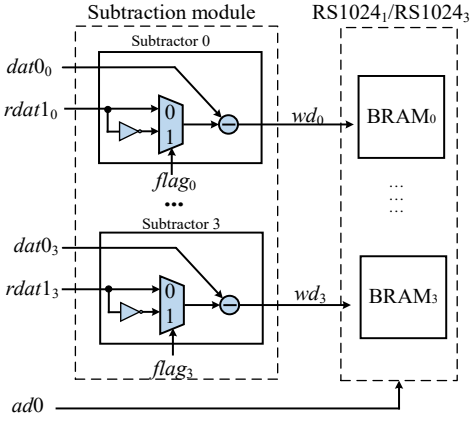


Fig. 16. The design of subtraction module.

Here, a simple example is used to illustrate the overall process in Phase 1. Suppose α_i=1027 and i=128. Four coefficients of d_0 have indices as (256, 768, 257, 769). Meanwhile, the indices of four coefficients of d_1 can be computed as (253, 765, 254, 766) by OAG. They are located in $BRAM_2$, $BRAM_3$, $BRAM_0$ and $BRAM_1$, respectively. The DR unit guarantees that when the four coefficients of d_0 are read from $BRAM_0$ to $BRAM_3$, their counterparts in d_1 can be read from $BRAM_2$, $BRAM_3$, $BRAM_0$ and $BRAM_1$, respectively. The coefficients of d_1 will be inversed or maintained based on the inverse flag from IFG. Finally, subtraction is performed between four pairs of coefficients and the subtraction result is written back to the 128th unit of $RS1024_1$/$RS1024_3$.

Phase 2. Decomposition and transformation

After Phase 1, the generated coefficients should be decomposed into two (as l=2) new coefficients. Two decomposed coefficients are first transformed to the numbers in the Galois field and then transformed to the Montgomery form, as shown in Fig. 17. These transformations facilitate the NTT operation in the next phase. To improve the efficiency, eight coefficients from 8 BRAMs in two RS1024 are processed in parallel, such that it only takes about 256 clock cycles to complete the operation in this phase.

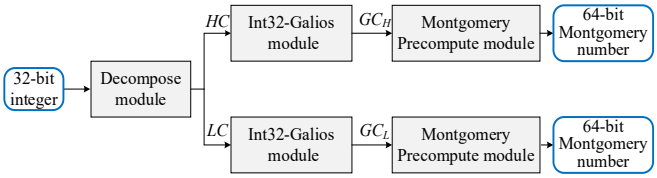


Fig.17. Decomposition and transformation in Phase 2.

As discussed in Section II, in TRGSW, l is set as 2 and two base numbers in H are set as B_1=2^{12} and B_2=2^{22}, respectively. Then, each coefficient C in $RS1024_1$/$RS1024_3$ should be decomposed into two new coefficients for the consequent processing. According to B_1 and B_2, the 10 bits from the 12th bit to the 21st bit of C is intercepted as the primary part of the 32-bit low coefficient, LC while the 10 bits from the 22nd bit to the 31st bit of C is intercepted as the primary part of the 32-bit high coefficient, HC. As the coefficient is a signed number, if the most significant bit (MSB) of LC is '1', it indicates a negative number as (LC-2^{10}) and HC should be increased by 1. If HC denotes a negative number with MSB as '1', both LC and HC should be negated. Otherwise, the original coefficient C is a positive number and two new coefficients will not change. After decomposition, two coefficients derived from C in $RS1024_1$/$RS1024_3$ are assigned in $RS1024_0$/$RS1024_1$ and $RS1024_2$/$RS1024_3$, respectively.

Each decomposed 32-bit coefficient DC (HC or LC) is converted to a Galois number GC according to eqn. (9), where the 64-bit M={{32{1'b1}}, {31{1'b0}}, 1'b1}.

$$GC = DC \bmod M = \begin{cases} DC & DC \geq 0 \\ DC + M, & DC < 0 \end{cases} \tag{9}$$

Each coefficient GC in the Galois field is transformed to that in the Montgomery form by computing $GC \cdot r$ (mod m), where r=2^{32}. It was found in the Nufhe library [30] that for two given 32-bit integers, a, b, $[(a+b\cdot r)\cdot r] \bmod (r^2-r+1) = (a\cdot r-a-b) \bmod (r^2-r+1)$. Thus, m can be set as (r^2-r+1) and, the Montgomery form of GC can then be efficiently computed by eqn. (10).

$$\mathfrak{M}(GC) = a \cdot r - a - b \;\; with \; a = GC[31:0], b = GC[63:32] \tag{10}$$

Phase 3: Computation of external product

In this phase, the operation of external product is performed between a TRGSW sample and a TRLWE sample. The TRLWE sample is denoted by a polynomial vector with $(k+1)l$=4 polynomials while the TRGSW sample denotes the bootstrapping key as a $(k+1)l\times(k+1)$=4×2 polynomial matrix.

It is noted that the complexity of the multiplication between two N-item polynomials is $O(N^2)$. To accelerate the polynomial multiplication, the NTT with negative wrapped convolution, abbreviated as NCN [26], is adopted. The multiplication between two polynomials can be completed in the following 5 steps, as shown in Algorithm 2.

Step 1. Two polynomials in $\mathbb{T}_N[X]$, $A(X)$ and $B(X)$ are preprocessed with the N coefficients of each polynomial being

multiplied by ψ^i, $i\in[0, N-1]$ to obtain $A_\psi(X)$ and $B_\psi(X)$, respectively. Here, ψ denotes the $2N^{th}$ root of unit in $\mathbb{T}$. This multiplication is called coefficient-wise multiplication and denoted by $\odot$. Its complexity is only $O(N)$.

Step 2. NTT is applied on $A_\psi(X)$ and $B_\psi(X)$ and they will be transformed to the polynomials in NTT domain as $\mathcal{A}(X)$ and $\mathcal{B}(X)$. The complexity of this transformation is $O(N\log_2 N)$.

Step 3. The multiplication between $\mathcal{A}(X)$ and $\mathcal{B}(X)$ can be completed using the coefficient-wise multiplication to generate $\mathcal{C}(X)$.

Step 4. $\mathcal{C}(X)$ in NTT domain is then transformed to that in the normal domain as $C_\psi(X)$ using Inverse NTT (INTT), which simply replaces the N-th root of unit ω_N present in NTT with ω_N^{-1}. Similarly, the complexity of INTT is $O(N\log_2 N)$.

Step 5. Coefficient-wise multiplication is performed between $C_\psi(X)$ and ψ^{-i}, $i\in[0, N-1]$ to compute $C(X)$.

Thus, with the application of NTT, the complexity of the external product reduces from $O(N^2)$ to $O(N\log_2 N)$.

Algorithm 2: NTT with negative wrapped convolution (NCN)

Input: Polynomial $A(X)\in\mathbb{T}_N[X]$, $B(X)\in\mathbb{T}_N[X]$.

Input: N-th root of unit $\omega_N\in\mathbb{T}$, $2N$-th root of unit $\psi\in\mathbb{T}$.

Output: Polynomial $C(X) = A(X)\times B(X)\in\mathbb{T}_N[X]$

1. $A_\psi(X) = A(X)\odot(\psi^0, \psi^1, \ldots, \psi^{N-1})$, $B_\psi(X) = B(X)\odot(\psi^0, \psi^1, \ldots, \psi^{N-1})$
2. $\mathcal{A}(X) = \mathbf{NTT}(A_\psi(X), \omega_N)$, $\mathcal{B}(X) = \mathbf{NTT}(B_\psi(X), \omega_N)$
3. $\mathcal{C}(X) = \mathcal{A}(X)\odot\mathcal{B}(X)$
4. $C_\psi(X) = \mathbf{NTT}(\mathcal{C}(X), \omega_N^{-1})$
5. $C(X) = C_\psi(X)\odot(\psi^0, \psi^{-1}, \ldots, \psi^{-(N-1)})$

As follows, the hardware implementation of each step in external product is elaborated.

1). Pre-processing

In pre-processing, N twiddle factors as $\psi^0, \psi^1, \ldots, \psi^{N-1}$ are multiplied to N coefficients, respectively. Two address generators, as shown in Fig. 18, are used to generate consistent addresses for the ROM of twiddle factors and BRAMs of coefficients, respectively. Then, each pair of factor and coefficient is multiplied and the result will be used to update the BRAM for the input coefficient.

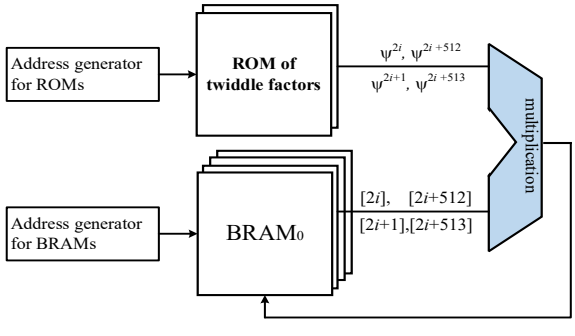
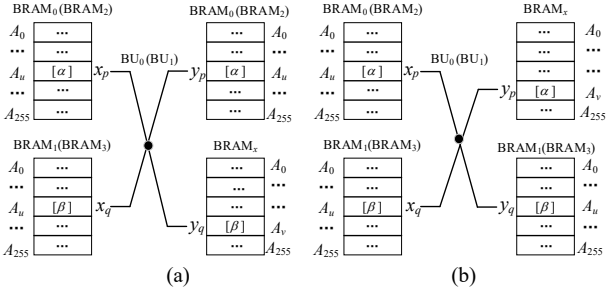


Fig. 18. Hardware of pre-processing for NTT.

2) NTT

NTT is then applied to transform two polynomials $A_\psi(X)$ and $B_\psi(X)$ as $\mathcal{A}(X)$ and $\mathcal{B}(X)$ in the NTT domain. To improve the efficiency of NTT in reading and writing, each dual-port BRAM storing the polynomial coefficient is configured with one port for reading and the other for writing so that BRAM can operate in pipeline mode. To facilitate reading, two inputs to BU_0/BU_1, x_p and x_q, are fetched from the memory with the equivalent address A_u in two paired BRAMs, $BRAM_0/BRAM_2$ and $BRAM_1/BRAM_3$, as shown in Fig. 19. As it is not necessary to write two outputs of a BU to the units where two inputs are read, we can assign one output to the memory unit where its corresponding input is read while the other output is written to the desired memory to facilitate the reading in the next stage. For example, the output y_q is written to the memory with address A_v in $BRAM_0/BRAM_2$, as shown in Fig. 19(a). The output y_p is written to the memory with address A_v in one $BRAM_x$ ($x\in[0, 3]$), as shown in Fig. 19(b).



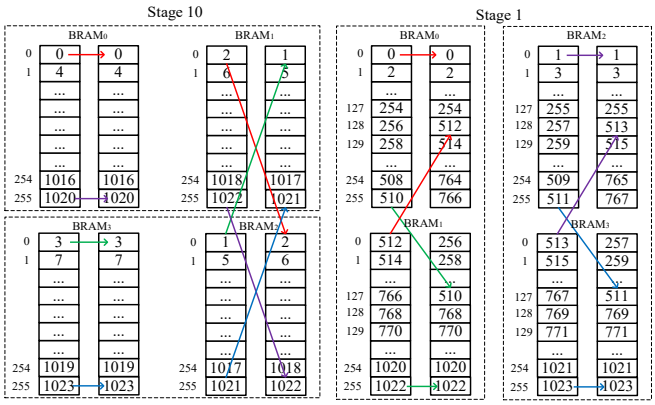
(a) y_q is written to a different position. (b) y_p is written to a different position.

Fig. 19. Reading and writing in BU_0 and BU_1.

Suppose x_p and x_q are fetched from the memory A_u in $BRAM_a$ and $BRAM_b$ ($a\neq b$), respectively. The assignment of two outputs from a BU to BRAMs can be classified into two cases.

Case 1: Two outputs of a BU are assigned to two different BRAMs.

In this case, y_p and y_q are written to the memory A_u in $BRAM_a$ and A_v in $BRAM_b$, respectively. For example, as shown in Fig. 20(a), in Stage 10 of 1024 points NTT, the input butterfly pair ([0], [2]) after each operation are written to $BRAM_0$ and $BRAM_2$, as shown by the red arrows. Another input butterfly pair ([1], [3]) are written to $BRAM_1$ and $BRAM_3$, as shown by the green arrows. Similarly, all other output butterfly pairs in Stage 10 are assigned in two BRAMs.These assignments are consistent with the configured BRAM(one port for reading and one port for writing).



(a) Case 1: 2 outputs to 2 BRAMs. (b) Case 2: 2 outputs to one BRAMs.

Fig. 20. The read-write strategies in two cases.

Case 2: Two outputs from a BU are assigned to one BRAM uniformly.

In this case, y_p and y_q are written to the memory A_u and A_v in BRAM$_a$, respectively. For example, as shown in Fig. 20(b), in Stage 1, the input butterfly pair ([0], [512]) from BRAM$_0$ and BRAM$_1$ are both written to BRAM$_0$ after each butterfly operation, as shown by the red arrows. However, as only one port of BRAM is used for writing, two outputs cannot be accommodated to one BRAM simultaneously. Thus, y_p can be written to A_u in BRAM$_a$ immediately while y_q is delayed for one clock cycle to be written to A_v in BRAM$_a$. For the example in Fig. 20(b), at clk$_0$, ([0], [512]) are read from BRAM$_0$ and BRAM$_1$, respectively. The output [0] is written in BRAM$_0$ at clk$_1$ while the other output [512] is written in BRAM$_0$ at clk$_2$. It is also noted that after y_q is written to the memory A_v in BRAM$_a$, the data in A_v of BRAM$_a$, [β] is erased and can never be read. Thus, [β] should be read prior to y_q being written to A_v. While y_p is written to A_u, the data [β] and its paired data are read simultaneously. y_q is written to A_v in the next clock. For the example in Fig. 21, at clk$_0$, ([0], [512]) are read from A_0 in BRAM$_0$ and BRAM$_1$, respectively. At clk$_1$, the output [0] is written to A_0 in BRAM$_0$ and [256] in A_{128} of BRAM$_0$ and [768] in A_{128} of BRAM$_1$ are read. At clk$_2$, the other output [512] is written to A_{128} in BRAM$_0$.

With this strategy, both outputs of a BU can be written to one BRAM while each BRAM can keep working in pipeline mode with one port for reading and one for writing.

In the hardware implementation of 1024 point **NTT**, as shown in Fig. 22, two specific address generators respectively provide the addresses for RS1024 and ROM. With given addresses, the polynomial coefficients from the BRAMs in RS1024 and the corresponding factors of (ω_N^0, ω_N^1.... ω_N^{N-1}) in ROM are selected and output to BU. The BU is implemented as shown by the circuit in the right part of Fig. 22. BU can process two inputs to generate the result of NTT in the current stage.

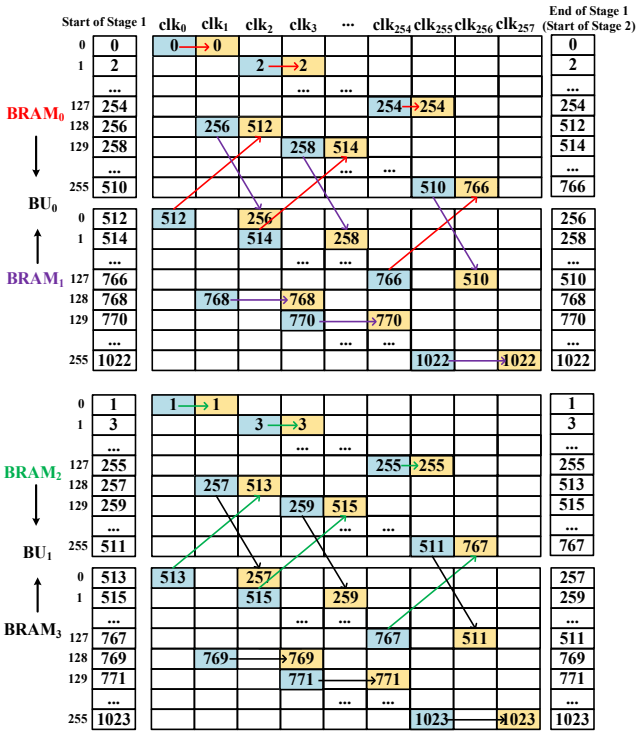


Fig. 21 Memory access for in Stage 1 for 1024 point NTT.

For the **INTT** operation, the ROM in Fig. 22 stores INTT factors ω_N^{-1} instead of ω_N to act as root of unity. The NTT/INTT operation for each polynomial are performed in parallel. Thus, both NTT and INTT operations cost similar time.

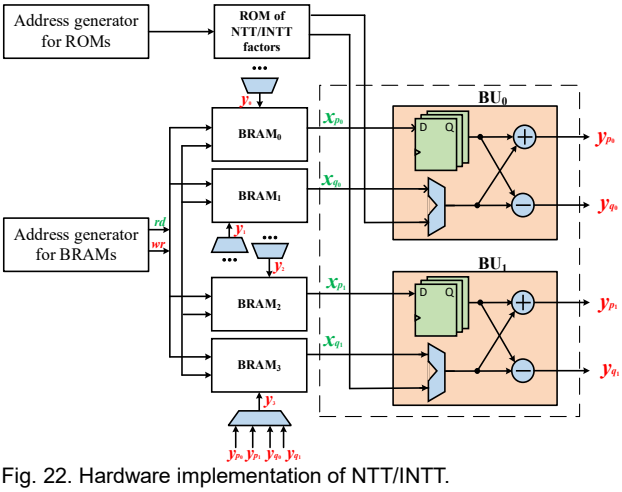


Fig. 22. Hardware implementation of NTT/INTT.

3) External product in NTT field

After the NTT operation, the TRLWE sample turns to a vector with four polynomials in NTT form stored in RS1024$_0$ to RS1024$_3$, respectively. Also, the bootstrapping key, as a TRGSW sample, has been transformed to a 4×2 polynomial matrix with each component polynomial in the NTT form. The external product is then performed between them as (5) to generate the output as a 1×2 polynomial vector. Each polynomial in the vector is finally written back to RS1024$_0$ and RS1024$_2$, respectively.

The product between the 1×4 TRLWE sample and the 4×2 TRGSW sample indicates 8 operations of polynomial multiplication. Thus, 8 public Montgomery multipliers are adopted to implement these 8 similar operations in parallel, as shown in Fig. 23.

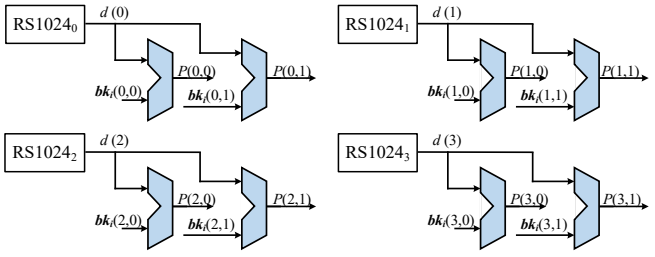


Fig. 23. 8 Montgomery multipliers for external product.

In each clock cycle, 4 coefficients of four polynomials in TRLWE sample, $d(m)$, $m \in [0..3]$, are multiplied with those of the first column $bk_i(m, 0)$ and the second column $bk_i(m, 1)$ in the TRGSW sample, respectively. After multiplication, the four products in each column $P(m, n)$, $n \in [0..1]$ are summed, as shown in Fig. 24. A subtractor is appended with each adder so that the summation result is guaranteed to be a Galois number.

Step 4 and Step 5 can be performed similarly as Step 2 and Step 1, respectively.

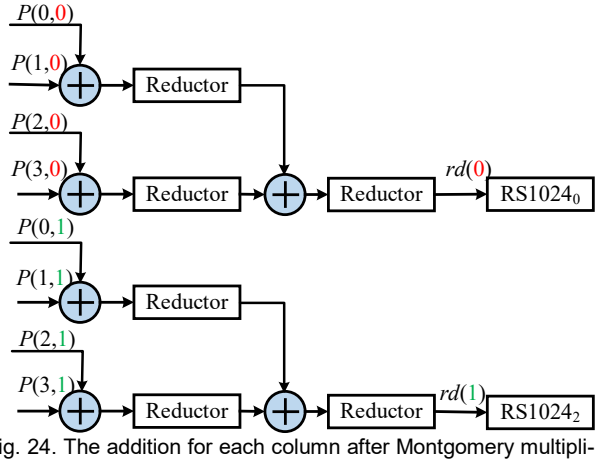


Fig. 24. The addition for each column after Montgomery multiplication.

Phase 4: Addition

In this phase, d_0 needs to be added to the resultant polynomials after obtaining the external product. As the subtraction between d_1 and d_0 in an earlier phase would overwrite the stored d_0, two extra RS1024 units are adopted to duplicate two polynomials in d_0 and update them each time d_0 is updated.

As the last step, the addition can be performed prior to the resultant polynomials being written to BRAMs so as to save at least 256 clock cycles. Thus, after the coefficient in Galois form is transformed to an int32, the corresponding coefficients of d_0 are fetched to the adder and the summed result is written to the BRAMs as the final result.

With all iterations of CMUX completed, the addition result from the last iteration will be read back to DDR through DMA, and the related software will receive the final bootstrapping result.

Thus, the proposed scheme explores how to minimize the latency in one bootstrapping operation by RTL optimization instead of pipelining or streaming the data path. With more resources and higher bandwidth memory available, pipelining or streaming can be adopted to further optimize the data path, hence the efficiency.

4 Evaluation and Analysis

In the experiment, the proposed scheme is performed on the ZYNQ UltraScale+MPSoC ZCU102 from the Xilinx company. This ZYNQ board consists of the Process Unit (PS) and the Programmable Logic (PL). The PS contains several ARM cores and multiple peripherals. PL denotes the FPGA of ZCU102. In the proposed design, PS is used to provide input data to bootstrapping and control the data transmission and interaction while PL acts as the dedicated processor to accelerate bootstrapping. The synthesis is implemented using Vivado 2018.3 and SDK. 8. As discussed in Section 2, the parameter set we used was: N=1024, k=1, n=500, l=2, B_g=1024. The data of other schemes using a similar parameter set are included for a fair comparison with the proposed scheme.

As the efficiency is of utmost importance, the performance in theory is first evaluated based on simulation and the practical performance is then measured. The operation in each phase is simulated and the consumed clock cycles are presented in Table II. It can be seen that the operations in the first two phases cost around 550 clock cycles, which accounts for 6.15% of the overall time. This shows that the proposed scheme on the organization of memory and hardware design contributes much to the improvement of the efficiency. For the operation of external product in Phase 3, the adoption of NCN reduces its complexity from $O(N^2)$ to $O(N\log_2 N)$ and this operation requires around 8,500 clock cycles. As the operation of addition in Phase 4 is merged into Phase 3, it takes zero time for the last phase. Overall, the operations in one loop for bootstrapping costs 9,066 clock cycles. If the FPGA works at the frequency of 300MHz, then 500 loops in the overall bootstrapping will take $9066\times500/(3\times10^8)$=15.11ms.

During the experiment, the modules on the board are well scheduled so that the bootstrapping key is ready prior to the start of each round of operation. The operations in the FPGA are never suspended due to the data communication among multiple modules. Hence, the practical time for gate bootstrapping should be equivalent to that in theory. The practical time for bootstrapping is measured to be 15.112ms using the timer in ARM. The slight difference of 0.002ms between the theoretical time and the practical time may be caused by the inherent error of the timer and it can be neglected. The practical time is hence consistent with the theoretical one. As 8 different ciphertexts are processed in the overall process, the bootstrapping on each ciphertext costs 1.9ms on average.

In Table III, the proposed scheme is compared to other schemes of gate bootstrapping using different processors with respect to performance and tradeoffs. As the performance is usually approximately proportional to the frequency of the processor frequency, we hence introduce the

Tradeoff Factor, $TF=1/fL$, where f denotes the clock frequency and L denotes the latency for bootstrapping. Greater TF indicates a better tradeoff. To compare other schemes with the proposed one, the *Tradeoff Factor Ratio*, $TFR= TF_x/TF_p$ is adopted, where TF_x and TF_p denote the TF of each compared scheme x and the proposed scheme, respectively. When the ratio is smaller than 1, it shows that the proposed scheme can achieve better tradeoff. Besides the latency, the throughput TP is also a factor to demonstrate the performance of gate bootstrapping. It can be denoted by the number of bootstrappings per second. Similarly, we introduce the *Throughput Factor*, $TPF=TP/f$ where large TPF denotes a good tradeoff between the throughput and resource consumption.

Table II. The Number of Clock Cycles in Each operation by Simulation.

Phase		Operation	# Cycle	Percentage
Phase 1		Rotation & subtraction	285	3.14%
Phase 2		Decomposition	273	3.01%
Phase 3	Step 1	Pre-processing	602	93.85%
	Step 2	NTT	3268	
	Step 3	External product	1069	
	Step 4	INTT	3268	
	Step 5	Post-processing	301	
Phase 4		Addition	0	0%
Total		--	9066	100%

Table III. Comparison of Performances among existing Bootstrapping Schemes on Different Processors.

Schemes		Processors	Frequency (f: GHz)	Latency (L: ms)	$TF(10^{-8})$	TFR	TP	TPF (10^{-9})
TFHE[14]		I7-7700HQ	2.9	13	2.6	0.01	50	17.2
TFHE-rs[35]		Intel Xeon Silver 4208[22]	2.1	9.45	5.04	0.03	110	52.38
cuFHE	Github[16]	NVIDIA Titan XP graphics card	1.4	0.5	143	0.82	--#	--#
	YKP[20]	NVIDIA GeForce RTX 3090	1.7	9.34	6.3	0.04	9579	5635
	DAFHE[36]	NVIDIA GEFORCE GTX 1050 Ti	1.3	2.9	26.5	0.15	--#	--#
MATCHA[23]		ASIC	2.0	0.6	83.3	0.48	3500	1750
Prop.		ZCU102	0.3	1.9	175	1	526	1753

#The data on throughput are not available in the reference.

The compared schemes include TFHE, TFHE-rs and cuFHE, which are implemented using a CPU, GPU and ASIC, respectively. The "TFHE" in Table III denotes the most popular TFHE gate bootstrapping library proposed in [13]. Using a CPU with 2.9GHz frequency, the scheme in [14] can implement gate bootstrapping [13] on a ciphertext in 13ms. TFHE-rs [35] denotes a pure Rust implementation of TFHE for Boolean and integer arithmetics over encrypted data. Using an Intel Xeon Silver 4208, it can implement THFE in 9.45ms with a 2.1GHz clock. The cuFHE library in [16] has been implemented with different GPU platforms, as shown in Table III. The github repository [16] shows that cuFHE can be implemented efficiently in 0.5ms using the NVIDIA Titan XP graphics card. MATCHA [23] implements a TFHE accelerator as an ASIC. It can achieve 0.6ms latency.

Our proposed scheme, denoted by "Prop.", can implement bootstrapping in 1.9ms on FPGA with 300MHz frequency. It can achieve a 6.8x, 5.0x, 4.9x and 1.5x speedup compared to the schemes using a CPU and GPU in [14], [35], [20] and [36], respectively. Only the implementation by [16] and the ASIC in [23] are faster than the proposed scheme. It is noted that their clocks are 4.7X and 6.7X higher than ours. If under similar clock frequency, then we can achieve 1.23X and 2.11X speedup compared to [16] and [23]. It can be seen that our proposed scheme can achieve the best tradeoff and it is more efficient than most acceleration schemes by other processors in terms of the bootstrapping latency. However, the cuFHE implemented in [20] can achieve the best throughput.

The proposed scheme is also compared with other published work on FPGA-based bootstrapping acceleration in Table IV. The column 'platform' shows the applied FPGA development board and processing core if any. The column 'Design level' denotes whether the acceleration-oriented optimization is implemented during HLS or at RTL. The consumed resource is enumerated in the column 'resource consumption'. To facilitate the discussion on the tradeoff between performance and consumed resources, the logic blocks of 'LUT', 'FF' and 'DSP' inside the FPGA are synthesized to find that they are roughly equivalent to 6.25, 4.75 and 4,690 2-input NAND gates in terms of the synthesized design area. Thus, the consumed logic resource can be uniformly represented by the number of equivalent NAND gates, G. The column 'Memory' denotes the quantity of the consumed 'BRAM', M. The column 'Clock' shows the frequency of the working clock f in MHz. The column 'Latency' denotes the average time to bootstrap one ciphertext, L in ms. The column 'TP' denotes the throughput.

Table IV. Comparison among FPGA-based Schemes

Scheme	platform	Design level	Resource consumption				Clock (f: MHz)	Latency (L: ms)	TF (10^{-14})	TFR	TP	TPF (10^{-12})
			Logic(G)			Mem(M)						
			#LUT (6.25)	#FF (4.75)	#DSP (4690)	#BRAM						
HSCD [18]	Zedboard	HLS	30K	10K	186	3.7	50	3401	2.95	1.2E-2	--#	--#
			1107.3K									
SPSL [19]	Zedboard		36K	24K	40	2.9	50*	17640	0.82	3.3E-3	0.057	8.3E-3
			526.5K									
YKP [20]	VU13P		842k	662k	7202	338	180	3.76	23.1	9.2E-2	3454	3.0
			42184.4K									
NOMP [21]	AlveoU280 i7-6700	RTL + software on CPU	521k	659k	4096	35.9	250	2.43	93.3	3.7E-1	3980	9.02
			25596.7K									
FPT [22]	Alveo U280	RTL, streaming processor	595k	1024k	5980	14.5	200	0.58	681	2.7	25000	98.7
			36629.0K									
Prop.	ZCU102	RTL	216k	238k	1024	12.9	300	1.9	252.1	1	526	2.52
			7283.06k									

*The clock frequency is conservatively selected the minimum one in all the schemes.

#The data on throughput are not available in the reference.

A shorter latency indicates a scheme can achieve better performance in bootstrapping. However, both higher clock frequency and more resource consumption contribute to a lower latency. Therefore, besides the latency, the tradeoff between the performance and the consumed resources can better exhibit the performance of the scheme itself. We similarly introduce the *Tradeoff Factor*, *TF*. The latency is usually a reciprocal of the clock frequency. The memory plays a more important role in improving the parallelism, hence the performance. Also, more logic resources indicate more potential to shorten the latency. *TF* is then defined as: $TF=1/fLM^{1/2}G^{1/3}$. Greater *TF* indicates a better tradeoff. To compare other schemes with the proposed one, the *Tradeoff Factor Ratio*, *TFR* is similarly defined as TF_x/TF_p, where TF_x and TF_p denote the *TF* of each compared scheme x and the proposed scheme, respectively. When the ratio is greater than 1, it shows the compared scheme can achieve a better tradeoff. Otherwise, the proposed scheme wins. Similarly, $TPF = TP/fM^{1/2}G^{1/3}$ and a large *TPF* denotes a good tradeoff.

The HSCD [18] only implements the complicated external product in FPGA with all other operations completed by software. A global optimization cannot be achieved among all operations and the switch between SW and HW results in a heavy time cost. Thus, its latency is more than three seconds and this low efficiency also results in a worse tradeoff. In SPSL [19], neither the optimization of the polynomial multiplication nor parallel computing is considered. Its latency is hence the largest at 17.6 seconds. As the working frequency is not provided [19], we conservatively use the minimum frequency among the compared schemes in Table IV, 50MHz as that in [19]. The worst latency by SPSL also makes it achieve the most unsatisfactory tradeoff. The scheme YPK in [20] proposed several parameterized bootstrapping primitives to enable more parallelism and optimization. Also, the key unrolling scheme is used to achieve different tradeoff flexibly. The latency under different configurations can all be reduced to less than 8ms. Here, the result which can achieve the best tradeoff between latency and throughput is selected for comparison. The latency is just 3.76ms while it consumes more memory and resource, which compromises its tradeoff. The NOMP in [21] introduces TGCs to improve the parallelism while using CPU to schedule TGCs. The latency is achieved as 2.43ms, and the consumed FPGA resources are less than [20], hence achieving a better tradeoff.

The FPT [22] exploits the inherent noise in FHE computation to accelerate the most complex operation FFT and uses the streaming processor to facilitate the cascading and pipeling in bootstrapping. The performance is improved remarkably and the latency is reduced to 0.58ms, although it consumes more memory and resources.

We can see that the *Latency* performance of the proposed implementation outperforms the schemes in [18]-[21] by 1790X, 9284X, 1.98X and 1.28X, respectively. As our proposed scheme consumes less resources, our tradeoff is better than those in [18]-[21]. We reason this is because the acceleration schemes in [18]-[20] are implemented usingHLS. The performance of the circuits synthesized by HLS is usually less optimized as the optimization potential cannot be fully explored. Also, the deep pipelining as a result of HLS usually requires more resource and memory. RTL provides more potential for the global optimization. The proposed scheme fully exploits the potential and implements all the optimized design at RTL. Thus, it achieves better performance and a better tradeoff than most of other schemes [18]-[21]. FPT [22] successfully improves the efficiency of the most complicated operation and adopts an efficient architecture with a streaming processor, which makes it achieve the best performance and tradeoff.

In terms of the performance of throughput, three recent YKP [20], NOMP [21] and FPT [22] schemes outweigh the proposed scheme. This analysis guides us to combine the specific architecture or optimization techniques with our

RTL optimization for a further improvement of the bootstrapping efficiency in the future work.

5 Conclusion

Nowadays, people extensively rely on the services from cloud to implement complex computation. However, this results in serious threats to the security and personal privacy. Fully homomorphic encryption is hence developed to enable arbitrary computation directly on ciphertext while the user can recover the decrypted computation result without revealing private data. FHE can solve the problem of data security, however, the operation of bootstrapping in FHE is more complex, which affects the computational efficiency of FHE.

Bootstrapping has been implemented using general-purpose processors, FPGAs and ASICs. We propose to use an FPGA to accelerate the bootstrapping in TFHE with deep RTL customization and optimization. The gate bootstrapping can be divided into four phases of operation on polynomials as rotation and subtraction, decomposition, external product and addition. To improve the efficiency of the key operation external product, each input ciphertext in N-stage polynomial form is stored in four dual-port BRAMs. In the first phase, with a specific memory organization, the coefficients of the origin polynomial and the rotated polynomial can be read simultaneously and $2N$ operations can be implemented in parallel in $N/4$ clock cycles. The decomposition is implemented by three connected fully-pipelined modules in $N/4$ cycles with low latency. NTT is adopted to facilitate the operation of external product. By transformation of the original polynomials to NTT form and specific memory access scheme, the complexity of external product can be reduced from $O(N^2)$ to $O(N\log_2 N)$. The last operation of addition is merged into the 3rd phase so as to save more time.

The proposed scheme is implemented on the FPGA chip ZCU102 on the ZYNQ board. With FPGA clocked at 300MHz, we can achieve an average latency of 1.9ms per ciphertext. Compared with the fastest GPU or CPU acceleration scheme TFHE-rs [35], even under a lower frequency, the proposed scheme can achieve 5.0X speedup. If using a similar clock frequency, our scheme will be 1.23X and 2.11X faster than the cuFHE [16] by GPU and TFHE by ASIC [23], respectively. Compared with most other implementations on FPGAs in terms of latency, the proposed scheme can also achieve better efficiency and tradeoff.

REFERENCES

[1] Y. Shu, L. Ailin, "Overview of the development of privacy-preserving computing," *Information and Communications Technology and Policy*, vol. 47, no. 6, 2021, pp. 1-11.

[2] Seo. M, "Fair and secure multi-party computation with cheater detection," *Cryptography*, vol. 5, no.3, 2021, pp. 19.

[3] K. Suzaki, K. Nakajima, T. Oi and A. Tsukamoto, "Library implementation and performance analysis of GlobalPlatform TEE internal API for Intel SGX and RISC-V keystone," in *Proc. IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Guang Zhou, China, Dec. 2020, pp. 1200-1208.

[4] M. Iezzi, "Practical pivacy-preserving data science with homomorphic encryption: an overview," in *Proc. IEEE International Conference on Big Data (Big Data)*, Nov. 2020, pp. 3979-3988.

[5] R. L. Rivest, L. Adleman, M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation*, vol. 76, no. 4, 1977, pp. 169-179.

[6] R. L. Rives, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, 1978, pp. 120-126.

[7] P. Paillier P, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Advances in Cryptology—EUROCRYPTO 1999*, Santa Barbara, California, USA, Aug. 1999, pp. 223-238.

[8] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. The 41st Annual ACM Symposium on Theory of computing*, Bethesda, Maryland, USA, May. 2009, pp. 169-178.

[9] C. Gentry, Z. Brakerski, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. The 3rd Innovations in Theoretical Computer Science Conference*, Cambridge, MA, USA, Jan. 2012, pp. 309-325.

[10] J. Fan, F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, 2012, pp. 144.

[11] J. H. Cheon, A. Kim, M. Kim, et al, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Advances in Cryptology—ASIACRYPT 2017, Part I*. Hong Kong, China, Dec. 2017, pp. 407-437.

[12] I. Chillotti, N. Gama, M. Georgieva, et al, "Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds," in *Proc. Advances in Cryptology—EUROCRYPT 2015, Part I*, Sofia, Bulgaria, April. 2015, pp. 617-640.

[13] (Apr. 2021). TFHE, [online]. Available: https://github.com/tfhe/tfhe.

[14] I. Chillotti, N. Gama, M. Georgieva, et al, "Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE," in *Proc. Advances in Cryptology—ASIACRYPT 2017, Part I*, Hong Kong, China, Dec. 2017, pp. 377-408.

[15] T. P. Zhou, X. Y. Yang, L. F. Liu, "Faster bootstrapping with multiple addends," *IEEE Access*, vol. 6, 2018, pp. 49868-49876.

[16] W. Dai, "CUDA-accelerated fully homomorphic encryption library," https://github.com/vernamlab/cuFHE.

[17] J. L. Hennessy, D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, 2019, pp. 48-60.

[18] B. Bulut, S. N. Bicakci, I. San, "HW/SW Co-Design of TFHE Homomorphic OR gate via NTT-based Polynomial Multiplication on programmable SoC," in *Proc. 2022 Innovations in Intelligent Systems and Applications Conference (ASYU)*, Antalya, Turkey, 2022.

[19] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieus, 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, pp. 1-7.

[20] T. Ye, R. Kannan and V. K. Prasanna, "FPGA Acceleration of Fully Homomorphic Encryption over the Torus," in *Proc. 2022 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2022, pp. 1-7.

[21] K. Nam, H. Oh, H. Moon and Y. Paek, "Accelerating N-bit Operations over TFHE on Commodity CPU-FPGA", in *Proc. 2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, San Diego, CA, USA, 2022, pp. 1-9.

[22] M. V. Beirendonck, J.-P. D'Anvers, and I. Verbauwhede, "FPT: A fixed-point accelerator for Torus fully homomorphic encryption," in *Proc the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2023, pp. 741–755.

[23] L. Jiang, Q. Lou, and N. Joshi, "MATCHA: A fast and energy efficient accelerator for fully homomorphic encryption over the torus," in *DAC'22: 59th ACM/IEEE Design Automation Conference*, San Francisco, California, USA, July 2022, pp. 10-14.

[24] J. W. Cooley, J. W. Tookey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19. no. 90, 1965, pp. 297-301.

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

[26] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, vol. 71, no. 11, 2022, pp. 2829-2843.

[27] I. Chillotti, N. Gama, M. Georgieva, et al, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, 2020, pp. 34-91.

[28] O. Regev. "On lattices, learning with errors, random linear codes, and cryptography" *Proceeding of the 31th annual ACM symposium on Theory of computing*, vol. 56, no. 6, 2005. pp. 34.

[29] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. Advances in Cryptology — CRYPTO 2013*, Santa Barbara, CA, USA, Aug. 2013, pp. 75-92.

[30] NuCypher. "NuCypher fully homomorphic encryption (NuFHE) library implemented in python," https://github.com/nucypher/nufhe.

[31] T. Morshed, M. M. A. Aziz and N. Mohammed, "CPU and GPU accelerated fully homomorphic encryption," in *Proc. 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, San Jose, CA, USA, May. 2020, pp. 142-153.

[32] T. Poppelmann, T. Guneysu, "Towards efficient arithmetic for lattice-based cryptography and reconfigurable hardware," in *Proc. Progress in Cryptology—LATINCRYPT 2012*, Santiago, Chile, Oct. 2012, pp. 139-158.

[33] W. Liu, F. Pan, X. Yang, T. Zhou, "Debug and analysis of the FHE software library based on GPU", *Netinfo Security*, vol. 6, 2019, pp. 76-83.

[34] P. Longa, and Naehrig. M, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Proc. International Conference on Cryptology and Network Security*, Houston, TX, USA, Oct. 2016, pp. 124-139.

[35] https://docs.zama.ai/tfhe-rs/getting-started/benchmarks

[36] W. Liu, F. Pan, X. Yang, and T. Zhou, "Debug and analysis of fully homomorphic encryption library based on GPU", *Netinfo Security*, 2019, 19(6), pp. 76-83

Jian Zhang received the Bachelor of sciense from Shantou University, Shantou, China, in 2020, and received the Master of engineer from Harbin Institute of Technology (Shenzhen), Shenzhen, China, in 2023. His current research interests include fully homomorphic encryption and hardware acceleration.

Aijiao Cui (S'05–M'10-SM'20) received the B.Eng. and M.Eng. degrees in electronics from Beijing Normal University, Beijing, China, in 2000 and 2003, respectively, and the Ph.D. degree in electrical and electronic engineering from Nanyang Technological University, Singapore, in 2009. From July 2003 to December 2004, she was a Lecturer with Beijing Jiaotong University, Beijing. She was a Research Fellow with Peking University Shenzhen SoC Laboratory, Shenzhen, from 2009 to 2010 prior to joining in the Harbin Institute of Technology (Shenzhen) in 2010, where she is currently a Professor in the School of Integrated Circuits. Her current research interests include hardware security and trusted IC design.

Yier Jin (Senior Member, IEEE) is a professor in the University of Science and Technology of China. He is also a courtesy professor in the Department of Electrical and Computer Engineering in the University of Florida. He received his PhD degree in Electrical Engineering in 2012 from Yale University after he got the B.S. and M.S. degrees in Electrical Engineering from Zhejiang University, China, in 2005 and 2007, respectively.

His research focuses on the areas of hardware security, embedded systems design and security, trusted hardware intellectual property cores and hardware-software co-design for modern computing systems. He is also interested in the security analysis on Internet of Things and wearable devices with particular emphasis on information integrity and privacy protection in the IoT era. Dr. Jin is a recipient of the DoE Early CAREER Award in 2016 and ONR Young Investigator Award in 2019.