FPGA Acceleration of Fully Homomorphic Encryption over the Torus

Tian Ye
Department of Computer Science
University of Southern California
Los Angeles, USA
tye69227@usc.edu

Rajgopal Kannan
US Army Research Lab
Los Angeles, USA
rajgopal.kannan.civ@mail.mil

Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, USA
prasanna@usc.edu

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926381

Abstract—Fully Homomorphic Encryption over the Torus (TFHE) is a promising approach for secure computing in cloud servers to perform computations directly on encrypted data. However, TFHE has much higher computation complexity than its unencrypted counterpart.

In this work, we propose an FPGA accelerator for TFHE computations. We illustrate the effects of an optimization called bootstrapping key unrolling on the tradeoff between performance of bootstrapping and FPGA resource consumption. We customize the data layout of TFHE ciphertext to optimize data access and improve data reuse. We parameterize the FPGA design for TFHE bootstrapping, which can be configured to achieve high performance for different user-specified security requirements and given FPGA resources. We implement our design on a state-of-the-art FPGA and compare it with existing results on CPUs. Our implementation for TFHE bootstrapping achieves 216× improvement in throughput and 16.5× improvement in latency compared with the software baseline on a state-of-the-art CPU server.

Index Terms—Privacy-Preserving Computation, Fully Homomorphic Encryption, FPGA

I. Introduction

Data security and privacy have always been major concerns in cloud computing. Despite the fact that many encryptions can protect the confidential data in transit, the data are still exposed to cloud servers, which are not always trusted by users. Fully Homomorphic Encryption (FHE) [1] addresses this issue by allowing the servers to directly perform computations over the encrypted data. The data is encrypted by the users locally into ciphertext before being sent to the cloud servers. The servers perform computations directly on the encrypted data (without decrypting the data) and return the result, still in ciphertext, back to the users for decryption. No one other than the users has access to the plaintext data, ensuring end-to-end privacy throughout the protocol.

For most FHE schemes (e.g, BFV [2], BGV [3], CKKS [4]), the depth of computations over the ciphertext is limited unless the users decrypt and re-encrypt the ciphertext or an expensive procedure called bootstrapping [5] is triggered. Additionally, those FHE schemes are incapable of performing non-linear computations (e.g., ReLU) and must approximate them using polynomial functions, which significantly reduces the accuracy. Fully Homomorphic Encryption over the Torus (TFHE) [6] is a variant that supports non-linear computations of any depth due to its faster bootstrapping procedure. As a result, TFHE is well suited for applications such as high-accuracy deep machine learning inference and training.

However, a critical disadvantage of TFHE is its high computational complexity in comparison to its unencrypted counterpart that exposes data in plaintext to the servers. FPGAs are appropriate accelerators for efficient TFHE implementations because they enable customized architectures that take use of the potential parallelism among the various TFHE primitives. Challenges arise when designing an efficient FPGA accelerator for TFHE, including: (1) As the TFHE scheme has various possible security levels, the design should be able to support a wide range of security levels instead of being customized only for specific parameters (e.g., security level, ciphertext size). (2) Because FPGAs have limited on-chip memory and the TFHE ciphertext is significantly larger than plaintext, data layout and data reuse should be optimized to reduce on-chip memory access conflicts and utilize the limited external memory bandwidth.

In this paper, we propose an efficient FPGA accelerator for the TFHE bootstrapping primitive. In the proposed architecture, we customize the data layout of TFHE ciphertext to optimize the data access and improve the data reuse. We propose parameterized IP cores for TFHE primitives and their subroutines, which can be configured to achieve high throughput for TFHE primitives for different security parameters. Our main contributions include:

- We design the first FPGA architecture for TFHE primitives. To enable efficient multi-level parallelism, we customize the data layout of TFHE ciphertext for FPGA on-chip SRAM to optimize data access and reduce memory access conflict. Our data layout also improves data reuse to effectively utilize the external memory bandwidth.
- To exploit the optimization called bootstrapping key unrolling, we evaluate its effects on the performance of TFHE bootstrapping. Our design is parameterized and can be configured to achieve high throughput and low latency for TFHE bootstrapping for different user-specified TFHE security requirements.
- We conduct detailed experiments to evaluate the proposed design. For TFHE bootstrapping, our implementation achieves 216× improvement in throughput and 16.5×

improvement in latency compared with the CPU baseline [7].

II. Background

A. Fully Homomorphic Encryption over the Torus

Fully Homomorphic Encryption (FHE) provides a solution to secure cloud computing by allowing the servers to directly perform computations over ciphertext. Multiple variants of FHE schemes have been proposed, e.g., BGV [3], BFV [2], [8], CKKS [4]. For all those schemes, the ciphertext has a noise term that rapidly grows along with homomorphic multiplications and would make the plaintext unretrievable after a specific number of computations. The "bootstrapping" procedure of those schemes, which can refresh the noise term to a minimal level, have prohibitively high computation complexity. Hence, those schemes only support applications with limited depth of computations.

Fully Homomorphic Encryption over the Torus (TFHE) [6] is a recent scheme that has much faster bootstrapping procedure and thus makes it practical to compute computations with arbitrary depth. Here we describe the basic preliminary knowledge about TFHE needed in this paper. The TFHE scheme consists of three types of ciphertext:

LWE ciphertext. Let q be a modulus (typically, $q = 2^{32}$ or 2^{64}). The plaintext space is $\mathbb{Z}_q$, meaning integers modulo q. A secret key for LWE ciphertext is n bits denoted as $s = (s_1, \ldots, s_n) \in \{0,1\}^n$. Under the secret key s, a plaintext integer $m \in \mathbb{Z}_q$ is encrypted into a LWE ciphertext $LWE(m) = c = (a_1, \ldots, a_n, b)$ where $a_1, \ldots, a_n$ are n random integers sampled from $\mathbb{Z}_q$ and $b = a \cdot s + m + e = \sum_{i=1}^{n} a_i s_i + m + e \pmod q$ with a small noise term e.

RLWE ciphertext. The plaintext space is $\mathbb{Z}_{q,N}[X]$, mean-ing a polynomial of degree $N-1$ with all coefficients in $\mathbb{Z}_q$. A secret key for RLWE ciphertext is a polynomial $S = \sum_{i=0}^{N-1} S_i X^i$ where $S_i \in \{0,1\}$. Under the secret key $S(X)$, a plaintext polynomial $M(X)$ is encrypted into a RLWE ciphertext $RLWE(M(X)) = (A(X), B(X))$ where $A(X)$ is randomly sampled from $\mathbb{Z}_{q,N}[X]$ and $B(X) = A(X) \cdot S(X) + E(X) + M(X)$ with a small-coefficient noise polynomial $E(X)$.

RGSW ciphertext. RGSW and RLWE share the same plaintext space and secret key. The RGSW ciphertext is in $\mathbb{Z}_{q,N}[X]^{2L\times 2}$, meaning a matrix with $2L$ rows and each row is a RLWE ciphertext. A subroutine required in this paper is the external product $\boxdot : RGSW \times RLWE \rightarrow RLWE$. It receives as input an RGSW ciphertext $RGSW(m_1)$ and an RLWE ciphertext $RLWE(m_2)$, where m_1 and m_2 are plaintext polynomials, and outputs an RLWE ciphertext $RLWE(m_1 \cdot m_2)$. The external product consists of $4L$ polynomial multiplications and additions.

Usage of LWE, RLWE and RGSW ciphertext. Typically, LWE ciphertexts are used to encrypt every integer or real number from the input data. Linear computations, including addition and constant multiplication, can be performed on LWE ciphertexts. RLWE and RGSW ciphertexts are usually used to encrypt constant values required in the programmable bootstrapping procedure, which is described as follows.

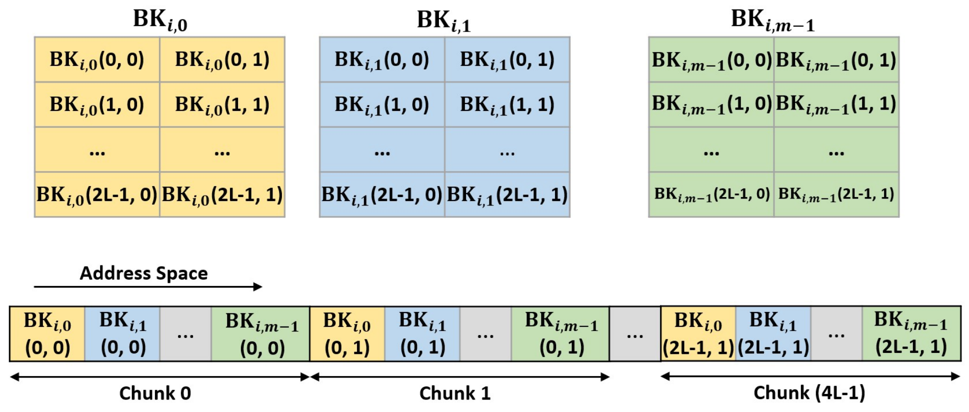
Programmable bootstrapping (PBS). PBS is the most important procedure in TFHE that can simultaneously reduce the noise term of a LWE ciphertext and compute an arbitrary non-linear function. The algorithm of PBS is shown in Algorithm 1. The input is the LWE ciphertext c_{in} encrypting a plaintext m. The output is the LWE ciphertext c_{out} encrypting the plaintext $f(m)$ with low noise, which is equivalent to apply an arbitrary function $f : \mathbb{Z}_q \rightarrow \mathbb{Z}_q$ to the underlying plaintext. The RLWE ciphertext c_T is considered as an input-independent constant encoding the function f. An RGSW ciphertext BK_i encrypts a bit s_i of the secret key s. Line 3-7 is to homomorphically compute $c_T \cdot X^{-b+c\cdot s} = c_T \cdot X^{-b+\sum_{i=1}^{n} a_i s_i}$. All the polynomial multiplications in Line 3, 5 and 6 will be computed using NTT/INTT that reduces the complexity from $O(N^2)$ to $O(N \log N)$. Due to the limit of space, we refer the readers to [6] and [7] for more details.

Algorithm 1: Programmable Bootstrapping (PBS)

```
Input: LWE ciphertext $c_{in} \in \mathbb{Z}_q^{n+1}$
Const: RLWE ciphertext $c_T \in \mathbb{Z}_{q,N}[X]^2$;
       RGSW bootstrapping key $BK_i$ $(i = 1, \ldots, n)$;
       KeySwitching key $KSK_{i,j}$ $(i = 1, \ldots, N,$
       $j = 1, \ldots, l_{ks})$
Output: LWE ciphertext $c_{out} \in \mathbb{Z}_q^{n+1}$
1  $c \leftarrow \lfloor \frac{2N}{q} \cdot c_{in} \rceil$
2  $(a_1, \ldots, a_n, b) \leftarrow c$
3  $ACC \leftarrow X^{-b} \cdot c_T$
4  for ($i \leftarrow 1$; $i \leq n$; $i \leftarrow i + 1$) do
5      $tmp \leftarrow (X^{a_i} - 1) \cdot BK_i + 1$
6      $ACC \leftarrow tmp \boxdot ACC$
7  end
8  $(a'_1, -a'_N, -a'_{N-1}, \ldots, -a'_2, b') \leftarrow ACC$
9  for ($i \leftarrow 1$; $i \leq n$; $i \leftarrow i + 1$) do
10     Decompose $a'_i$ into $a'_{i,1}, \ldots, a'_{i,l_{ks}}$
11 end
12 $c_{out} \leftarrow (0, \ldots, 0, b') - \sum_{i=1}^{N} \sum_{j=1}^{l_{ks}} a'_{i,j} \cdot KSK_{i,j}$
```

B. Secure Cloud Computing Protocol

In our secure cloud computing protocol, the server is defined as the cloud provider who offers computing service, and the client is defined as the owner of confidential data who wants to use the computing service from the server. Initially, the client and server makes an agreement on the encryption parameters (e.g., security levels and polynomial degree). The client generates private keys and sends necessary constants (e.g., bootstrapping key and keyswitching key) to the server. Then the client encrypts the private data locally, and sends the ciphertext to the server. The server receives the input data and starts to perform computation over the ciphertext directly. Since TFHE can support non-linear and arbitrary depth of computations, all computations can be entirely handled by the server alone. After finishing all computations, the server sends



$BK_{i,0}$
$BK_{i,0}(0, 0)$ $BK_{i,0}(0, 1)$
$BK_{i,0}(1, 0)$ $BK_{i,0}(1, 1)$
…
$BK_{i,0}(2L-1, 0)$ $BK_{i,0}(2L-1, 1)$
$BK_{i,1}$
$BK_{i,1}(0, 0)$ $BK_{i,1}(0, 1)$
$BK_{i,1}(1, 0)$ $BK_{i,1}(1, 1)$
…
$BK_{i,1}(2L-1, 0)$ $BK_{i,1}(2L-1, 1)$
$BK_{i,m-1}$
$BK_{i,m-1}(0, 0)$ $BK_{i,m-1}(0, 1)$
$BK_{i,m-1}(1, 0)$ $BK_{i,m-1}(1, 1)$
…
$BK_{i,m-1}(2L-1, 0)$ $BK_{i,m-1}(2L-1, 1)$
Address Space
$BK_{i,0}$ (0, 0) $BK_{i,1}$ (0, 0) … $BK_{i,m-1}$ (0, 0)
$BK_{i,0}$ (0, 1) $BK_{i,1}$ (0, 1) … $BK_{i,m-1}$ (0, 1)
…
$BK_{i,0}$ (2L-1, 1) $BK_{i,1}$ (2L-1, 1) … $BK_{i,m-1}$ (2L-1, 1)
Chunk 0
Chunk 1
Chunk (4L-1)

Fig. 1: Bootstrapping key layout in DRAM

the output, still in ciphertext, back to the client. The client decrypts it into the final output. In the entire process, no others except the client can have access to the private data, and thus it guarantees end-to-end security for the cloud computing.

III. Related Work

Homomorphic Encrypted Neural Network Inference: Most prior works on secure neural network inference use FHE schemes such as BFV [2], BGV [3] and CKKS [4] that do not support non-linear activation functions. To compute non-linear functions, a class of works [9], [10] used Server-client protocol, which exactly computed activation functions by combining FHE and Multi-Party Computation. While this method is friendly to deep computations, it requires the client to communicate with the server frequently and perform significantly more computations, which is contrary to the intention of cloud computing, i.e., exporting the client's heavy computations to the server. Another class of works [11]–[13] used Server-only protocol, where activation functions are approximated by the square function $f(z) = z^2$. However, this approach reduces the prediction accuracy and is only acceptable for neural networks with limited number of layers. In contrast, TFHE is a promising scheme for exact computation of non-linear functions entirely on the server without requiring the client to do much computation or communication. It also supports deep neural network inference and training without limitation on the network's depth.

Hardware Accelerations for TFHE: There have been a few previous works that accelerate TFHE using ASIC, GPU and FPGA. [14] provided acceleration for TFHE bootstrapping on ASIC that was optimized for a fixed design point. [15] accelerated TFHE primitives on GPUs. Compared with GPUs, FPGAs have the following advantages in accelerating TFHE: (1) FPGAs can be customized to exploit the parallelism across different subroutines of TFHE procedures, such as NTT and vector multiplication. (2) FPGAs enable manipulation of the data layout in the on-chip memory in order to reduce the latency associated with TFHE data access. (3) FPGAs consume less energy than GPUs, which is advantageous when deploying large number of instances of accelerators. [16] proposed a naive FPGA architecture for TFHE primitives. However, their design does not obtain superior performance compared with the software implementation on CPUs due to a lack of optimizations on polynomial multiplications and the algorithm of TFHE bootstrapping. In this paper, we focus on a more efficient FPGA architecture for TFHE primitives. Our design achieves high throughput and low latency for the TFHE's bootstrapping procedure.

IV. Accelerator Design

A. Parameterized Bootstrapping Key Unrolling

In Algorithm 1, the bottleneck is the loop of n iterations in Lines 4-7. In each iteration, ACC is homomorphically multiplied by $X^{a_i s_i} = (X^{a_i} - 1) \cdot s_i + 1$ where s_i is in the form of a RGSW ciphertext BK_i. Thus, the loop is equivalent to multiplying c_T by $X^{\sum_{i=1}^{n} a_i s_i}$. Typically, n is 500 to 1024 [7]. As the iterations have data dependency on the variable ACC, the n iterations must be computed sequentially. Bootstrapping key unrolling, first proposed in [17] and extended in [14], unrolls every m iterations and reduces the number of iterations into n/m, where m is the unrolling factor. For the example of $m = 2$, what to be multiplied by ACC in iteration i becomes $X^{a_{2i+1}s_{2i+1}+a_{2i+2}s_{2i+2}} = (X^{a_{2i+1}+a_{2i+2}} - 1) \cdot s_{2i+1}s_{2i+2} + (X^{a_{2i+2}} - 1) \cdot s_{2i+2}(1 - s_{2i+1}) + (X^{a_{2i+1}} - 1) \cdot s_{2i+1}(1 - s_{2i+2}) + 1$. Let $BK_{i,0}$, $BK_{i,1}$ and $BK_{i,2}$ be the RGSW ciphertext of $s_{2i+1}s_{2i+2}$, $s_{2i+2}(1 - s_{2i+1})$ and $s_{2i+1}(1 - s_{2i+2})$ respectively. Line 5 in Algorithm 1 will be modified as $tmp \leftarrow (X^{a_{2i+1}+a_{2i+2}} - 1) \cdot BK_{i,0} + (X^{a_{2i+2}} - 1) \cdot BK_{i,1} + (X^{a_{2i+1}} - 1) \cdot BK_{i,2} + 1$.

In our design, we let m be a configurable parameter to support bootstrapping key unrolling for an arbitrary unrolling factor m. Intuitively, a larger m requires higher computational complexity in Line 5 but reduces the number of iterations. Its effect on the performance (latency and throughput) will be demonstrated and analyzed in Section V.

B. Data Layout and Memory Access

The main data traffic between the external DRAM and on-chip SRAM is to load the bootstrapping keys

$BK_{i,0}, BK_{i,1}, \ldots, BK_{i,m-1}$. As shown in Figure 1, each bootstrapping key $BK_{i,j}$ is an RGSW ciphertext consisting of $2L \times 2$ polynomials of degree $(N-1)$. For efficient access from the external DRAM, the polynomials in the m bootstrapping keys are rearranged as illustrated at the bottom of Figure 1. The rearrangement can be done offline and has no overhead on the performance because the bootstrapping keys are dependent on the TFHE secret keys and irrelevant to the input ciphertext. The FPGA loads all the m polynomials belonging to one chunk from the DRAM and feeds a replica to the buffer at the input port of each Processing Element (PE). The Bootstrapping Key Unit (BKU) in each PE reads the polynomials from the buffer and performs the computations. Details about the BKU is described in Section IV-C. To limit the usage of on-chip SRAM, the buffers hold polynomials from only one chunk at a time. A new chunk of polynomials are loaded from the DRAM after the previous chunk has been consumed by the BKU.

C. Architecture for TFHE Bootstrapping

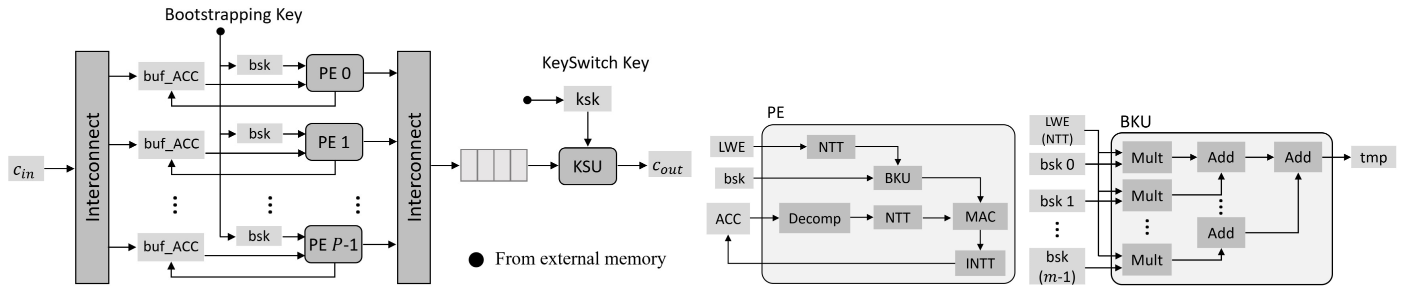
The overall architecture for TFHE bootstrapping is shown in Figure 2(a). The external DRAM stores bootstrapping keys and keyswitch keys that will be loaded to the FPGA at runtime.

Processing Element (PE). The architecture contains P processing elements (PEs), each of which computes Lines 4-7 of Algorithm 1 for one instance of input ciphertext. Here P is a configurable parameter. Multiple instances of ciphertext are processed by the PEs in parallel. Each PE consists of a Bootstrapping Key Unit (BKU) for the computations in Line 5, a Multiply-Accumulate (MAC) module, two Number Theoretic Transform (NTT) modules, one Inverse NTT (INTT) module and one Decomposition module. The MAC module is responsible to perform the polynomial multiplications and additions of the external product in Line 6. The Decomposition module is to perform bitwise decomposition for each $(\log q)$-bit integer into l_{ks} smaller integers required in the external product procedure in Line 6. All the modules in a PE operate as a pipeline for high throughput. The outputs of the PEs are pushed into a queue which are consumed by the KeySwitching Unit (KSU) to perform the computation in Line 12. Details of the NTT, INTT, BKU and KSU are described as follows.

NTT and INTT. NTT and INTT simplify the computational complexity of polynomial multiplications from $O(N^2)$ to $O(N \log N)$, where N is the degree of the polynomial. After both input polynomials are transformed by NTT (i.e., transformed into their evaluation space), the polynomial multiplication is simplified into coefficient-wise multiplication. In the external product (Line 6 of Algorithm 1), every polynomial from ACC will be multiplied by two polynomials from tmp. We allocate one NTT module for the polynomials from ACC. As will be described later, the polynomials from tmp are transformed by an NTT module at an earlier step. An NTT module is a $(\log N)$-stage pipeline where each stage has a configurable number of NTT cores. Each NTT core processes a pair of coefficients by computing a modular multiplication, addition and subtraction. Let p be the modulus for NTT, which should be a prime integer no less than the maximum coefficient q. To minimize the consumption of DSPs, we extend the reduction algorithm in [18] and implement modular multiplication with modulus of $p = 2^{33} - 2^{20} + 1$ when $q = 2^{32}$, as shown in Algorithm 2. The notation [:] represents the slice of an integer by the specified range of bits. Line 1-13 are to compute modular multiplication $r = x_{in} \cdot w \bmod p$. It requires only one 32-bit × 32-bit integer multiplication (Line 1) in each NTT core that consumes 3 DSPs on the Xilinx FPGA we use in Section V. INTT is the inverse transform of NTT and thus has a similar accelerator design. One INTT module is deployed after the MAC module and outputs polynomials of ACC to the next iteration.

```
Algorithm 2: NTT core with $p = 2^{33}-2^{20}+1$, $q = 2^{32}$
    Input: Coeffcients $x_{in} \in Z_q$, $y_{in} \in Z_q$
    Const: Twiddle factor $w \in Z_p$
    Output: Coeffcients $x_{out} \in Z_p$, $y_{out} \in Z_p$
1  $z \leftarrow x_{in} \cdot w$
2  $c \leftarrow z[63:59] + z[58:46] + z[45:33]$
3  $d \leftarrow z[63:59] + z[63:46] + z[63:33]$
4  $e \leftarrow c[14:13] + c[12:0]$
5  $f \leftarrow (e[13:13]+e[12:0]) \cdot 2^{20} - e[13:13] - c[14:13]$
6  $r \leftarrow f + z[32:0]$
7  if $r \geq p$ then
8      $r \leftarrow r - p$
9  end
10 $r \leftarrow r - d$
11 if $r < 0$ then
12     $r \leftarrow r + p$
13 end
14 $x_{out} \leftarrow y_{in} - r \bmod p$
15 $y_{out} \leftarrow y_{in} + r \bmod p$
```

Bootstrapping Key Unit (BKU). The BKU in each PE reads the polynomials in the buffer and computes one polynomial for tmp in Line 5. Figure 2(c) shows the architecture of a BKU. For each of the m bootstrapping keys $BK_{i,0}, BK_{i,1}, \ldots, BK_{i,m-1}$, there is a buffer at the input port of the BKU that can store a polynomial at a time. The BKU reads m polynomials from the buffers, and performs m monomial-polynomial multiplications and m polynomial subtractions in parallel. The monomial-polynomial multiplication $BK_{i,j}(i', j') \cdot (X^c - 1)$ could be implemented by cyclically shifting the coefficient array of $BK_{i,j}(i', j')$ by c slots and subtracting them by the original coefficients. Here, (i', j') indexes one of the $2L \times 2$ polynomials for RGSW ciphertext $BK_{i,j}$, and c is a variable depending on $\{a_{im}, a_{im-1}, \ldots, a_{im-m+1}\}$. However, as the values of $\{a_1, a_2, \ldots, a_n\}$ are not known until runtime, the FPGA SRAM storing the coefficients would suffer from I/O conflict which significantly impacts the performance. Instead, we use NTT to simplify the monomial-polynomial multiplication into coefficient-wise multiplication. As $BK_{i,j}(i', j')$ depends on $\{s_1, s_2, \ldots, s_n\}$ that are known constants known in advance, their NTT can be computed



(a) Architecture for bootstrapping (b) Processing element (PE) (c) Bootstrapping key unit (BKU)

Fig. 2: Overall architecture

offline and thus does not affect the runtime performance. After all m result polynomials are computed, an adder tree receives and sums them up to generate an polynomial belonging to tmp. Note that the computed polynomials are still in their evaluation space and can directly be the input of the external product (Line 6 of Algorithm 1) that also consists of polynomial multiplications.

KeySwitching Unit (KSU). The KSU performs the computations in Line 8-12, where each $KSK_{i,j}$ is a vector of n coefficients. This module iteratively loads each vector and performs constant multiplications and accumulations. As the computation load of KSU is much lower than the PEs, only one instance of KSU is deployed without affecting the overall throughput of bootstrapping.

V. Experimental Results

A. Experimental Setup

Our FPGA designs are synthesized using Xilinx Vitis HLS 2020.2 [20]. The experiments are conducted on Xilinx Virtex UltraScale+ VU13P FPGA [21]. It has 12288 DSPs, 3456K FFs, 1728K LUTs, 5376 instances of 18 Kb BRAMs and 1280 instances of 288 Kb UltraRAMs. The peak external memory bandwidth is 77 GB/s.

For the CPU baseline, we implement the TFHE bootstrapping using Concrete library [7]. The baseline is performed on a state-of-the-art server with an AMD Ryzen Threadripper 3990X CPU @ 2.90 GHz with 64 cores and 128 threads. The server has 256 GB DDR4 with 200 GB/s peak bandwidth to DRAM. We also include results from previous work [16] as FPGA baseline. The device used in our paper and all the baselines are summarized in Table I.

B. Comparison on the Performance of TFHE Bootstrapping

We implement our design for TFHE bootstrapping on two different sets of TFHE parameters:

(I) $n = 500, N = 1024, L = 2, Bg = 1024$
(II) $n = 592, N = 2048, L = 3, Bg = 128$

(I) is adopted from [6] with 110-bit security and also used in [14] for fair comparisons. (II) is selected as a large parameter set with 80-bit security. For both parameter sets, we conduct experiments for four different values of m as 1, 2, 3 and 4, and measure their performance (latency and throughput) and resource consumption (LUT, FF, DSP, on-chip SRAM). The results are shown in Table II.

Effect of unrolling factor m: The table demonstrates that for a fixed parameter set, the latency keeps decreasing as the unrolling factor m increases owing to the reduction in the number of loop iterations n/m required in TFHE bootstrapping. The maximum throughput defined as the number of TFHE bootstrappings finished per second is obtained at $m = 2$. It increases as m grows from 1 to 2 because fewer iterations are required. However, the throughput decreases when m grows from 2 to 4 because larger m requires more on-chip SRAM in each PE leading to fewer PEs that can be deployed using the limited amount of on-chip SRAM in the FPGA. The value of m should be selected depending on whether the latency or throughput is more critical for the given application.

Comparison with CPU baselines: The CPU baselines are based on the implementation of TFHE bootstrapping provided by the Concrete library [7]. As the library does not support bootstrapping key unrolling, the unrolling factor m is set to 1. For parameter set (I) and (II), our implementation ($m = 2$) achieves 216× and 96× better throughput with 16.5× and 8.4× improvement in latency.

Comparison with prior FPGA implementation: SPSL2021 [16] is the only prior work accelerating TFHE bootstrapping on FPGAs. As their purpose is to demonstrate an application of an FPGA-based programmable vector engine, their design is a preliminary architecture on a low-end FPGA with many opportunities for optimization. For example: (1) It does not use NTT or FFT for faster polynomial multiplications. (2) It does not consider algorithmic optimizations such as bootstrapping unrolling. (3) It does not use parallel computing architectures such as pipelines. Therefore, even if we normalize the performance by the amount of consumed resources, our implementation still significantly outperforms this baseline.

Comaprison with prior ASIC and GPU implementations: There are also a few prior works on hardware acceleration for TFHE bootstrapping. MATCHA [14] is an ASIC design simulated by a modeling framework with the assumption of a high HBM2 bandwidth (640 GB/s) and high clock rate (2 GHz). For parameter set (I) and $m = 3$, MATCHA achieves latency of 0.2 ms and throughput of ∼10K

TABLE I: Resources used by various designs

	This paper	CPU w/ Concrete [7]	GPU w/ cuFHE [19]	SPSL2021 [16]	MATCHA [14]
Computation Resources	Xilinx VU13P @ 180 MHz 12288 DSP slices	AMD Ryzen 3990X @ 2.90 GHz 64 cores and 128 threads	NVIDIA RTX 3090 1.70 GHz 10496 CUDA cores	Xilinx 7Z020 220 DSP slices	Simulated Architecture 2 GHz 36.96mm^2 area
Peak External Memory Bandwidth	77 GB/s (DDR4)	200 GB/s (DDR4)	1008 GB/s (GDDR6X)	77 GB/s (DDR4)	640 GB/s (HBM2)

TABLE II: Comparison of TFHE bootstrapping implementations

Work	Parameter Set	m	LUT / FF / DSP / On-chip SRAM	Clock (MHz)	Latency (ms)	Throughput (BS/sec)[a]
This paper	(I)	1	925K / 729K / 6240 / 319Mb	180	7.53	1993
		2	842K / 662K / 7202 / 338Mb	180	3.76	3454
		3	569K / 448K / 6640 / 383Mb	180	2.51	3188
		4	442K / 342K / 6910 / 409Mb	180	1.88	2657
This paper	(II)	1	931K / 728K / 6272 / 343Mb	180	19.13	732
		2	770K / 602K / 6446 / 400Mb	180	9.56	1150
		3	534K / 419K / 6034 / 429Mb	180	6.38	1098
		4	353K / 279K / 5656 / 422Mb	180	4.78	836
Concrete [7]	(I)	1	——	——	62.0	16
	(II)	1	——	——	80.3	12
cuFHE [19]	(I)	1	——	——	9.34	9579
SPSL2021 [16][b]	——	1	36K / 24K / 40 / 2.9Mb	——	17640	0.057
MATCHA [14]	(I)	1	——	2000	0.6	3500
		2	——	2000	0.3	6600
		3	——	2000	0.2	10000
		4	——	2000	0.2	10000

[a]Number of bootstrappings per second.
[b]The parameter set and clock rate are not available in [16].

bootstrappings per second. Note that the maximum external memory bandwidth of our FPGA is 77 GB/s. If a higher bandwidth is available, we would achieve better throughput by offloading a part of the on-chip SRAM to external DRAMs and allocating more instances of PEs. For the metric of throughput per bandwidth, our design for m = 3 achieves 2.65× speedup over MATCHA. The cuFHE library [19] provides a CUDA implementation of TFHE bootstrapping for m = 1. We measure its performance for parameter set (I) on an NVIDIA GeForce RTX 3090 GPU @ 1.70 GHz with 10496 CUDA cores and 1008 GB/s HBM2 bandwidth, which achieves a latency of 9.34 ms and a throughput of 9579 bootstrappings per second. For a fair comparison, we use our design for m = 1 which achieves 1.24× improvement in latency, which means a better performance in latency-sensitive scenarios. Although we have lower throughput than GPU, it should be noted that the peak performance of the GPU is 17.8 TFLOPS while our FPGA device has peak performance of 2.21 TFLOPS. Also, the thermal design power (TDP) of the GPU is 350W, while the FPGA board has a power consumption of □50W (estimated by Xilinx Power Estimator). Therefore, for the metric of energy efficiency (defined as throughput per unit power), our implementation on FPGA is 1.46× better than the GPU baseline.

VI. Conclusion

In this paper, we proposed the first efficient FPGA architecture for TFHE bootstrapping primitive. Our design can be configured to achieve high throughput and low latency of bootstrapping for different user-specified security requirements. Our implementation of TFHE bootstrapping acceleration achieved 216× speedup in throughput and 16.5× latency improvement compared with a state-of-the-art CPU baseline. In the future, we will explore the application of TFHE bootstrapping on privacy-preserving machine learning applications and their hardware acceleration.

Acknowledgement

This work has been sponsored by the U.S. National Science Foundation under grant number SaTC-2104264.

References

[1] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison et al., "Security of homomorphic encryption," HomomorphicEncryption. org, Redmond WA, Tech. Rep, 2017.
[2] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 309–325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," Cryptology ePrint Archive, Report 2016/421, 2016, https://eprint.iacr.org/2016/421.

[5] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," Cryptology ePrint Archive, Report 2018/153, 2018, https://ia.cr/2018/153.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene,` "Tfhe: Fast fully homomorphic encryption over the torus," Cryptology ePrint Archive, Report 2018/421, 2018, https://ia.cr/2018/421.

[7] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, vol. 15, 2020.

[8] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," Cryptology ePrint Archive, Report 2012/078, 2012, https://eprint.iacr.org/2012/078.

[9] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," in Proceedings of the 27th USENIX Conference on Security Symposium, ser. SEC'18. USA: USENIX Association, 2018, p. 1651–1668.

[10] B. Reagen, W. Choi, Y. Ko, V. Lee, G.-Y. Wei, H.-H. S. Lee, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," 2020.

[11] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in Proceedings of The 33rd International Conference on Machine Learning, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 201–210. [Online]. Available: http://proceedings.mlr.press/v48/gilad-bachrach16.html

[12] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," 2017.

[13] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus," Cryptology ePrint Archive, Report 2018/1056, 2018, https://eprint.iacr.org/2018/1056.

[14] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," 2022.

[15] T. Morshed, M. M. A. Aziz, and N. Mohammed, "Cpu and gpu accelerated fully homomorphic encryption," 2020.

[16] 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," SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop). [Online]. Available: https://par.nsf.gov/biblio/10282639

[17] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," Cryptology ePrint Archive, Report 2017/1114, 2017, https://ia.cr/2017/1114.

[18] T. Ye, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Fpga acceleration of number theoretic transform," in High Performance Computing, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 98–117.

[19] V. Group, "Cuda-accelerated fully homomorphic encryption library," https://github.com/vernamlab/cuFHE, 2019.

[20] Xilinx, "Vivado design suite user guide: High-level synthesis ug902 (v2020.1)," 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis

[21] Xilinx, "Ultrascale architecture and product data sheet: Overview (ds890)," 2021. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf