International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 14, No. 3, November 2025, pp. 659∼675
ISSN: 2089-4864, DOI: 10.11591/ijres.v14.i3.pp659-675

Hardware design for fast gate bootstrapping in fully homomorphic encryption over the Torus

Saru Vig[1], Ahmad Al Badawi[2], Mohd Faizal Yusof[2]
[1]Department of Cybersecurity, Institute for Infocomm Research, Astar, Singapore
[2]Department of Homeland Security, Rabdan Academy, Abu Dhabi, United Arab Emirates

Article Info

Article history:

Received Feb 26, 2024
Revised Sep 8, 2025
Accepted Oct 9, 2025

Keywords:

Acceleration
Field-programmable gate array
Fourier transform
Hardware design
Torus fully homomorphic encryption

ABSTRACT

Fully homomorphic encryption (FHE) is a promising solution for privacy-preserving computations, as it enables operations on encrypted data. Despite its potential, FHE is associated with high computational costs. As the theoretical foundations of FHE mature, mounting interest is focused towards hardware acceleration of established FHE schemes. In this work, we present a hardware implementation of the fast Fourier transform (FFT) tailored for polynomial multiplication and aimed at accelerating gate bootstrapping in Torus fully homomorphic encryption (TFHE) schemes. Our study includes an extensive design-space exploration at various implementation levels, leveraging parallel streaming data to reduce computational latency. We introduce a new algorithm to expedite modular polynomial multiplication using negative wrapped convolution. Our implementation, conducted on reconfigurable hardware, adheres to the default TFHE parameters with 1024-degree polynomials. The results demonstrate a significant performance enhancement, with improvements of up to 30-fold, depending on the FFT design parameters. Our work contributes to the ongoing efforts to optimize FHE, paving the way for more efficient and secure computations.

Corresponding Author:

Ahmad Al Badawi
Department of Homeland Security, Rabdan Academy
Abu Dhabi 22401, United Arab Emirates
Email: ahmad@u.nus.edu

1. INTRODUCTION

With the rise of cloud computing, concerns regarding the privacy of sensitive data processed by third-party services have been mounting. This has driven the development of various privacy-preserving techniques (PETs), including differential privacy (DP) [1], secure multi-party computation (SMPC) [2], trusted execution environments (TEEs) [3], and fully homomorphic encryption (FHE) [4].

Among these, FHE offers a promising advantage by enabling computation on encrypted data without decryption, guaranteeing privacy even from the cloud provider, as shown in Figure 1. FHE supports arithmetic operations like addition and multiplication on encrypted data, allowing for computations through untrusted third-party services without compromising the underlying information. This makes FHE particularly suitable for scenarios where sensitive data analysis is required while maintaining stringent privacy guarantees. In a typical FHE workflow, the user encrypts a private input vector before sending it to an untrusted server that performs computations homomorphically. The resulting encrypted output is then returned to the user for decryption, revealing the computed result in plaintext.

Existing FHE schemes typically produce and operate on noisy ciphertexts. The noise magnitude in-

creases as homomorphic operations are performed on the ciphertexts, with homomorphic multiplication resulting in relatively higher noise growth compared to homomorphic addition. The noise cannot grow indefinitely, or decryption will fail to recover the message. To enable arbitrary computations, the ciphertexts must be refreshed to reduce their noise content. This noise control mechanism, known as bootstrapping [4], distinguishes FHE schemes from partial ones. However, bootstrapping is computationally expensive, especially in word-based FHE schemes, and is considered the main performance bottleneck of these schemes [5].

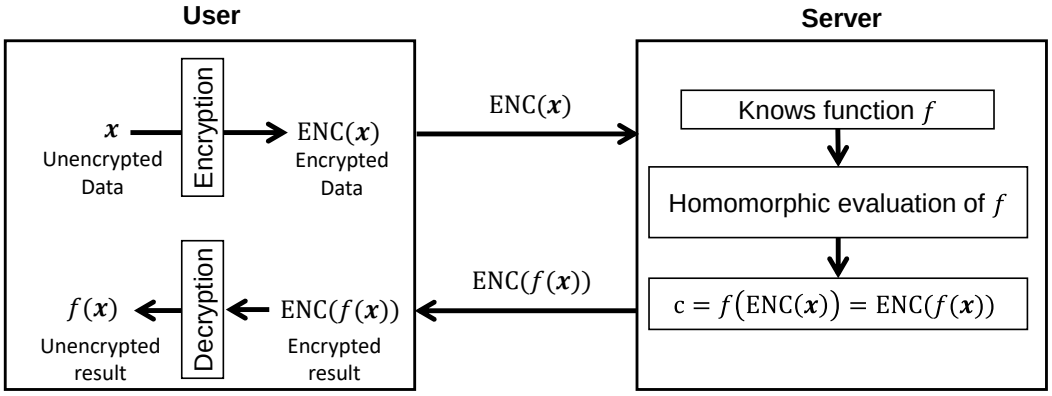

Figure 1. Overview of homomorphic computation in FHE

Torus fully homomorphic encryption (TFHE), which is an improved version of the FHEW scheme [6], stands out as a prominent FHE scheme recognized for its efficient bootstrapping procedure. Notably, it enables homomorphic computation of binary operations in a mere 13 milliseconds on a single core processor, utilizing a 16 MB bootstrapping key within the TFHE library [7]. While surpassing the performance of previous schemes, TFHE can be computationally expensive for arbitrary applications. This stems from its approach of encoding input data as individual bits or small-bit integers (typically limited to 8 bits) and representing functions as circuits of binary gates. However, its small parameter size and ability to compute a wide range of functions make it a promising candidate for future FHE applications.

We analyzed TFHE and identified FFT-based polynomial multiplication as the bottleneck operation during bootstrapping. This finding aligns with previous profiling of HE schemes [8], where it was reported that 75% of estimated cycles are spent in FFT convolutions used for polynomial multiplication. For TFHE, we calculated that the number of FFT operations required for a comparison function of two 16-bit numbers is approximately 3×10^6. This highlights the significant computational cost associated with FFT-based polynomial multiplication in TFHE and emphasizes the need for further research into more efficient approaches for bootstrapping in homomorphic encryption schemes.

Hardware implementations on field-programmable gate arrays (FPGAs) have demonstrated potential for significant performance gains in computationally intensive algorithms. This is particularly relevant to the field of homomorphic encryption, where applications like TFHE rely heavily on operations like the FFT. Recent work by Gener *et al*. [9] reinforces this notion by presenting an FPGA-based polynomial multiplication implemented as a vector-matrix product, achieving substantial acceleration compared to software implementations. Our work aims to leverage the inherent parallelism and configurability of FPGAs by exploring a hardware implementation of the FFT algorithm specifically tailored for TFHE-based applications. This approach has the potential to further enhance the efficiency and practicality of this family of homomorphic encryption schemes.

In this work, we present an optimized polynomial multiplication algorithm for TFHE schemes based on an extended variant of the negative-wrapped convolution (NWC) method. This well-established technique offers significant efficiency improvements for polynomial multiplication, a fundamental operation in FHE. Our proposed approach builds upon the NWC method by incorporating additional optimizations, aiming to further reduce the computational complexity and memory footprint associated with polynomial multiplication in FHE schemes.

We implemented and evaluated our polynomial multiplier utilizing the comprehensive suite of tools provided by Xilinx Vivado, specifically targeting the Virtex-7 xc7vx1140 FPGA. This included simulation,

synthesis, and hardware implementation. Our design achieves a significant performance gain, nearly 30 times faster compared to CPU software implementation, when using an FFT design of streaming width of 128 and radix-4 configuration. This configuration exhibits the optimal performance among the tested radix values.

Our contributions: the main contributions of our work are summarized as follows:

- We developed an efficient hardware architecture for the TFHE scheme on FPGAs, specifically targeting the time-consuming bootstrapping operation.
- We introduce a novel architecture for fast Fourier transform (FFT)-based polynomial multiplication. This innovative design significantly enhances the efficiency of polynomial multiplication, a critical component in TFHE bootstrapping.
- We have successfully leveraged our polynomial multiplier to accelerate the bootstrapping process in TFHE.
- We conducted a comprehensive performance evaluation of basic homomorphic binary gates using our accelerator achieving speedups close to 30× compared with CPU implementation.

Organization: the remainder of this paper is organized as follows. Section 2 introduces the methods, notations, and background concepts used throughout the paper. Section 3 then presents an analysis of the bottlenecks in the bootstrapping operation, along with our design space decisions, and concludes with a proof-of-concept implementation and evaluation of our register-transfer level (RTL) design. Section 4 discusses the results, while section 5 reviews related work on hardware acceleration. Finally, section 6 concludes the paper and highlights the key takeaways and future directions.

2. METHODS

This section details the methodologies employed throughout the paper. We begin by introducing the notation and symbols used, followed by a comprehensive description of the TFHE scheme, highlighting its core building blocks. Subsequently, we delve into the intricacies of the NWC approach, and finally, we introduce the SGen tool, a platform for automated design generation at the RTL.

2.1. Notations

Capital and lowercase letters distinguish sets from their elements. We denote the sets of binary, integer, real, and complex numbers by $\mathbb{B}$, $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{C}$, respectively. Matrices and vectors are represented by boldface capital and lowercase letters, respectively. The dot product between two vectors u and v is denoted by $\langle u \cdot u \rangle$. We use the symbols $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$, and $\lfloor \cdot \rceil$ to denote the floor, ceiling, and nearest integer functions, respectively. For an integer a, a_q denotes the remainder of a when divided by q. If a is a polynomial, the reduction is performed on each coefficient. $\mathbb{Z}_N[X]$ refers to the ring of polynomials $\mathbb{Z}/(X^N + 1)$. The symbol $a \leftarrow \mathcal{S}$ denotes sampling an element a from the set $\mathcal{S}$.

2.2. Torus fully homomorphic encryption

This section revisits the TFHE scheme, as introduced by Chillotti *et al*. [7]. Built upon the learning with errors (LWE) hardness problem [10], TFHE utilizes the Ring variant of LWE (RingLWE) [11]. Originally proposed as an optimized version of FHEW [6], the scheme facilitates homomorphic evaluation of binary operations. It also supports performing homomorphic arithmetic in $\mathbb{Z}_p$ with p being a low-width (typically ¡ 8-bit) integer modulus.

TFHE differentiates itself from earlier FHE generations by performing bootstrapping after each gate operation, a technique known as gate bootstrapping. This approach enables computations on commodity hardware, achieving speeds of less than 1 second for FHEW and around 0.13 seconds for TFHE. Subsequently, TFHE introduced the concept of circuit bootstrapping, where ciphertexts are refreshed after a series of gate evaluations. The key distinction between the two methods lies in the parameters size and execution of the bootstrapping procedure. For a detailed comparison, we refer the reader to [12].

2.2.1. Torus fully homomorphic encryption samples

TFHE operates with two distinct types of samples: Torus LWE (TLWE) and Torus GSW (TGSW). These samples play a crucial role in the TFHE scheme, each serving a distinct purpose in the encryption and homomorphic computation procedures. TLWE samples are primarily used for the encryption of individual bits, while TGSW samples are utilized for more complex operations such as bootstrapping and homomorphic computations. The interplay between these two types of samples is fundamental to the functionality and efficiency of the TFHE scheme. In the following paragraphs, we introduce these two mathematical notions.

The torus, denoted by $\mathbb{T}$, is defined as the quotient group $\mathbb{R}/\mathbb{Z}$, representing the real numbers modulo 1 under the standard addition operation. We further generalize this concept to $\mathbb{T}_q = (1/q)\mathbb{R}/\mathbb{Z}$, where q is a positive integer modulus. The construction of $\mathbb{T}_{N,q}[X]$ extends this definition to the space of polynomials whose coefficients reside in $\mathbb{T}_q$, with operations performed modulo $(X^N + 1)$ and modulo q. Additionally, $\mathbb{B}_N[X]$ is defined as a subset of polynomials in $\mathbb{Z}_N[X]$ where coefficients belong to a specific ring $\mathbb{B}$.

The scheme utilizes different types of samples, categorized as either TLWE and TGSW samples which are used within internal bootstrapping procedures, which we describe below:

TLWE Let $n, N \in \mathbb{N}$ denote the dimension and degree, respectively. Let $\mathcal{P} = \mathbb{T}$ be the plaintext space, the key space $\mathcal{S} = \{s_1, \ldots, s_n\} \in \mathbb{B}^n$, and the ciphertext space $\mathcal{C} = \mathbb{T}_q^{n+1}$. For a message $m \in \mathcal{P}$ (which can be a bit, modular integer, or real number in an interval), the encryption of m gives ciphertext sample c which takes the form $c = ((a_0, a_1, \ldots, a_{n-1}), b)$, where (1).

$$b = \sum_{i=0}^{n-1} a_i s_i + e + \Delta m. \tag{1}$$

Here, $a \leftarrow \mathbb{T}_q^n$ is sampled uniformly, **s** is the secret key, and e is the noise factor sampled from the underlying Gaussian distribution of TFHE. Decryption is performed as (2).

$$b - a \cdot s = \Delta m - e \xrightarrow{\text{Round}} m. \tag{2}$$

Building upon the descriptions above, TRLWE ciphertext samples can also be generated from polynomials. In this context, the plaintext space is denoted by $\mathbb{T}_{N,q}[X]$, the secret key vector by $S = \{s_1, ..., s_k\} \in \mathbb{B}_N[X]^k$, and the ciphertext by $C \in \mathbb{T}_{N,q}[X]^{n+1}$. Notably, TLWE is a specific instance of TRLWE with $N = 1$. TRLWE samples enable homomorphic addition and constant multiplication. The utilization of two distinct cipher schemes, namely TRLWE and TLWE, is primarily motivated by facilitating the internal bootstrapping process.

TGSW the Gentry–Sahai–Waters (GSW) scheme, a variant of the LWE encryption scheme, is capable of performing non-linear operations homomorphically. The TFHE scheme employs GSW for the homomorphic multiplication of two ciphertexts. This is particularly useful in bootstrapping, where an external product, denoted as $\boxdot$, is defined as TGSW $\times$ TLWE $\rightarrow$ TLWE. The primary application of the external product in TFHE is the controlled multiplexer, or CMUX. The operation of the CMUX is further defined later in the subsequent sections.

A TGSW sample can be conceptualized as a matrix of TLWE elements. TFHE employs a 'gadget decomposition' scheme to construct these matrices. Given the notations described above for the TLWE sample, the TGSW encryption scheme for a message $m \in \mathbb{Z}_p[X]$ is defined as (3).

$$C = Z + m \cdot H \tag{3}$$

Each row of Z is a homogeneous TLWE sample encrypted under key s. The gadget matrix, H, belongs to $\mathbb{T}_q^{(n+1)\times(n+1)l}$. Reciprocally, $C \in \mathbb{T}_q^{(n+1)l\times(n+1)}$ is considered a valid TGSW sample if there exists a message $m \in \mathbb{Z}_p[X]$ such that each row of $C - m \cdot H$ is a valid homogeneous TRLWE sample under the key. Both TLWE and TGSW schemes, along with their operations, can be extended to polynomials represented as TRLWE and TRGSW, respectively. The combination of these samples plays a crucial role in refreshing noisy ciphertexts during bootstrapping.

2.2.2. Gate level bootstrapping

Bootstrapping is a widely used technique to manage noise growth in homomorphic encryption schemes. It achieves this by homomorphically evaluating the decryption procedure on a ciphertext via an encryption of the secret key.

Here, we consider the example of a specific homomorphic encryption scheme, namely TFHE, which utilizes gate bootstrapping. This process takes a ciphertext of the form TLWE_μ, where μ is the plaintext message, as input. The output is another ciphertext, either encrypting 0 or μ, with a controlled amount of noise.

The gate bootstrapping procedure in TFHE consists of three key steps: i) BlindRotate, ii) SampleExtract, and iii) KeySwitch, which are described as follows.

– The BlindRotate operation takes a TLWE sample as input and multiplies it with a secret encrypted power of X, effectively producing a rotation of the coefficients. This rotation is achieved by invoking the CMUX gate in a loop for each coefficient of the cipher. A visual representation of BlindRotate can be found in Figure 2.

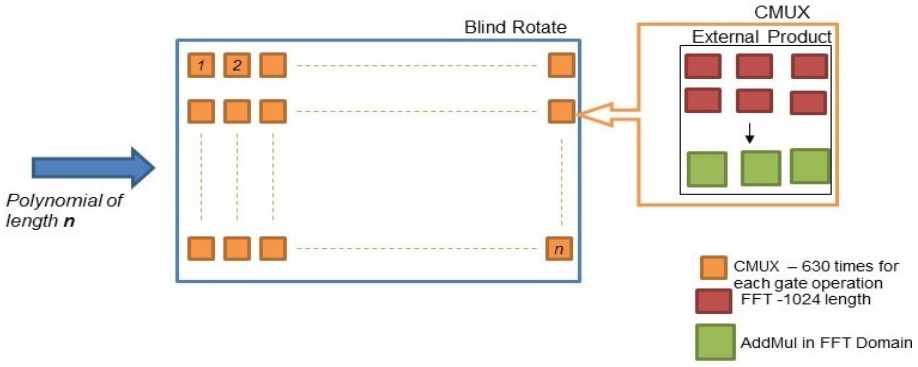

Figure 2. Overview of the BLINDROTATE operation performed during BOOTSTRAPPING after each gate-level operation

The procedure relies on polynomials, require shifting the input cipher sample from the TLWE($\mathbb{T}_N[X]$) space to the TRLWE space. Here, c denotes a TRLWE sample of a polynomial message $v \in \mathbb{T}_N[X]$ under the key $s = (s_1, ..., s_n) \in \mathbb{B}^n$, where the constant term is a rounded approximation of the original message in the $\mathbb{T}$ space.

Matching bootstrapping keys $s' = (s'_1, ..., s'_n) \in \mathbb{B}^n$ exist, with $C'_1, ..., C'_n$ representing their corresponding TGSW samples. The process begins by rounding up the input cipher and defining $\tilde{c}(\tilde{a_1}, ..., \tilde{a}_n, \tilde{b}) \leftarrow \lfloor c2N \rceil$ mod $2N$. The process is executed over two steps by defining an accumulator, ACC, as follows:

a. ACC $\leftarrow X^{-\tilde{b}} \cdot c$

b. Perform ACC $\leftarrow$ CMUX(C'_i,$X^{\tilde{a}_i} \cdot ACC$, ACC) for $1 \leq i \leq n$

The output is a TRLWE sample encryption of $X^{-p} \cdot v$, where $p = \tilde{b} - \sum_{i=1}^{n} s'_i \cdot \tilde{a}_i$ mod $2N$.

– The SampleExtract operation is the subsequent step following the conversion of plaintext v into the encryption of the polynomial $X^{-p} \cdot v$. In this operation, the constant term is extracted from the cipher. This extraction process results in the retrieval of the cipher of the original message, now encrypted under a new key. This operation is a crucial component in the key switching process, enabling the transition of encrypted data between different encryption keys.

– The KeySwitch operation is a fundamental component in homomorphic encryption schemes. This standard key switching algorithm is designed to convert a ciphertext encrypted under one key into a ciphertext encrypted under a different key. The implementation of this operation necessitates the use of key-switching keys. Specifically, these are the TLWE encryptions of the key bits of $\tilde{s}$, with respect to the original key s.

2.3. Negative wrapped convolution

Within the context of FHE, one of the computationally most expensive operations is the modular multiplication of polynomial elements. A prevalent approach to achieve this task efficiently leverages FFT-based convolutions. The cyclic convolution for two length-D signals x, y is denoted by a signal $z = x \times y$ having D elements as (4).

$$z_m = \sum_{i+j \equiv m(modD)} x_i y_j \tag{4}$$

Whereas the negacyclic convolution of x,y adds a factor of (-1) with $v = x \times_{\mathrm{m}} y$ and is denoted by [13]:

$$v_m = \sum_{i+j=m} x_i y_j - \sum_{i+j=D+m} x_i y_j \tag{5}$$

The acyclic convolution of the signal given by $u = x \times_{\mathrm{a}} y$ having $2D$ elements is given by:

$$u_m = \sum_{i+j=m} x_i y_j \tag{6}$$

for $m \in \{0, ..., 2D-2\}$. Lastly, the half-convolution is a length-D signal given by $x \times_{\mathrm{h}} y$ consisting of the first D elements of the cyclic convolution u. All the basic convolutions are closely related to one another. As shown in [13], if the length D is even and $x_j, y_j = 0$ for $j > D/2$,

$$L(x) \times_{\mathrm{a}} L(y) = x \times y = x \times_{\mathrm{m}} y \tag{7}$$

given the notion of the splitting of signals (of even length) into halves, the lower-indexed coefficients of a are denoted as $L(a)$. The above statement introduces us to the concept of "zero-padding" which is to append D zeros to signals of length D so that the signals' acyclic convolution is identical to the cyclic (or the negacyclic) convolution of the two padded sequences. While the classical cyclic convolution is applied to accelerated polynomial multiplication modulo X^N-1, negacyclic convolution is used to implement polynomial multiplication modulo X^N+1.

Now, that we have established the use of negacyclic convolution to perform modular multiplication, we explain the possible approach of zero-padding that has been implemented as part of the TFHE Library which uses the standard cyclic convolution and FFT to perform negacyclic convolution. We note that:

$$X^{2N} - 1 = (X^N + 1)(X^N - 1) \tag{8}$$

The goal is to obtain the product of two N-point polynomials modulo $(X^N + 1)$. First, the product modulo $(X^{2N} - 1)$ via cyclic convolution is computed, i.e., using the standard Fourier transform definition. To obtain the product modulo $(X^N + 1)$, the result needs to be reduced to the intermediate result modulo $X^N + 1$.

The description of how this reduction works was presented in [14] and is used here for completeness. Let us define a signal $p \in \mathbb{Z}[X]$ of degree $N-1$. Its negacyclic extension $\bar{p}$ (of length $2N$) is defined as (9):

$$\bar{p}[X] = p[X] - X^N \times p[x] \tag{9}$$

At each multiplication by X with the polynomial, the coefficients are circularly shifted one position to the right and the entering coefficient is negated. The resulting signal is such that its Fourier image will have zeros at all the even positions and the remaining coefficients will be mirrored and conjugated. Thus, multiplication of two such signals p and q can be performed as (10):

$$p \times q \mod (X^N + 1) = \frac{1}{2}\mathcal{F}^{-1}(\mathcal{F}(\bar{p}) \cdot \mathcal{F}(\bar{q}))[0...N-1] \tag{10}$$

Note that after $\mathcal{F}^{-1}$, the coefficients are negacyclic, hence we can only take the first half of the vector. This approach of negative wrapped convolution followed in the TFHE library is also described in Algorithm 1. Thus, the TFHE library uses a $2N$ length FFT architecture to perform multiplication. In our proposed approach, we use length N FFT architecture to achieve the same result for modular polynomial multiplication.

ALGORITHM 1: Polynomial multiplication using negative wrapped convolution

Input: $a(x), b(x) \in R^N$ are real valued polynomials of degree $(N-1)$
Output: $c(x) \in R^N$, $c = a * b \mod X^N + 1$

$\bar{a}$= $a(x) - x^N a(x)$
$\bar{b}$= $b(x) - x^N b(x)$
$\bar{A}$= $\mathcal{F}_{2\mathrm{N}}\, \bar{a}$
$\bar{B}$= $\mathcal{F}_{2\mathrm{N}}\, \bar{b}$
$\bar{C}$= $\bar{A} \cdot \bar{B}$
$\bar{c}$= ${\mathcal{F}^{-1}}_{2\mathrm{N}}\, \bar{C}$
for $j \leftarrow 0$ **to** $N-1$ **do**
  $c(j)$ = $\bar{c}(j)$
return c

2.4. Automatic generation of register-transfer level level design

An important step in gate bootstrapping is the generation of RTL designs for the involved functions. In this context, we explore the generalization of the hardware back-end of the SGen tool, presented in [15], to automatically generate data-path designs for FFTs.

SGen is an open-source tool that generates hardware designs for various signal processing transforms. It focuses on designs that operate on streaming data, where the input dataset is divided into smaller chunks processed over multiple cycles, leading to reduced resource usage. The FFT data path comprises $O(\log_2 N)$ cascaded stages for a signal of length N, as shown in Figure 3. Each stage processes w elements (words) per cycle, with a streaming width of w. Consequently, the number of cycles required to receive the entire input (or output) is $\frac{N}{w}$. The hardware in each stage is reused for subsequent sets of elements, effectively performing a "vertical folding" of the signal. SGen implements the Cooley-Tukey algorithm for streaming FFT and offers an area-efficient alternative to the work presented in [16] for complex data type FFT problems. A detailed comparison of these tools is presented later in this paper. Our work investigates the applicability of SGen as a tool for generating FFT data paths in the case of TFHE with 128-bit security level. We will leverage the data-path design generated by SGen and apply it to our proposed polynomial multiplication Algorithm 2.

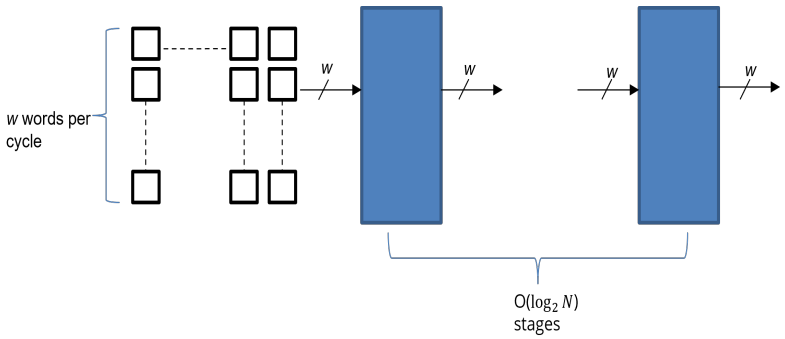

Figure 3. Streaming FFT architecture

ALGORITHM 2: Multiplication of N length real polynomial signals with $N/2$ length FFT

Input: $a(x), b(x) \in R^N$ are real valued polynomials of degree $(N-1), N = 2048$
Output: $c(x) \in R^N$, $c = a * b$

for $j \leftarrow 0$ **to** $N/2 - 1$ **do**
  $\hat{a}(j) = a(j) + a(N/2 + j)$ // fold
  $\hat{a}(j) = \hat{a}(j) \cdot \omega_N^j$ // twist
  $\hat{b}(j) = b(j) + b(N/2 + j)$
  $\hat{b}(j) = \hat{b}(j) \cdot \omega_N^j$
$\hat{A} = \mathcal{F}(\hat{a})$
$\hat{B} = \mathcal{F}(\hat{b})$
$\hat{C} = \hat{A} \cdot \hat{B}$
$\hat{c} = \mathcal{F}^{-1}(\hat{C})$
for $j \leftarrow 0$ **to** $N/2 - 1$ **do**
  $c(j) = \text{imag}(\hat{c}(j)) \cdot \omega_{2N}^{-j}$ // untwist and unfold
  $c(N/2 + j) = \text{real}(\hat{c}(j)) \cdot \omega_N^{-j}$;

3. IMPLEMENTATION

This section describes the fundamental building blocks of the bootstrapping approach implemented on the FPGA. Figure 4 provides a high-level overview of the entire architecture. Through complexity analysis, we identified the time-critical functions and targeted them for hardware acceleration.

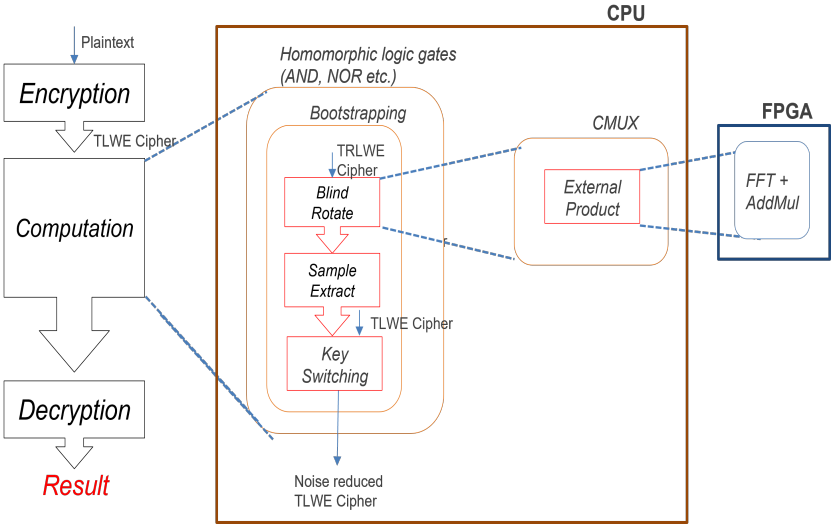

Figure 4. High-level system overview

The BLINDROTATE function is the core of the bootstrapping process, responsible for refreshing the noise. It performs a coefficient rotation by multiplying an encrypted polynomial with an encrypted power of X. This rotation is achieved using the CMUX gate, as illustrated in Figure 2. The CMUX gate takes two TLWE samples, d_0 and d_1, as inputs, along with a control TGSW sample, C. Its output is a TLWE sample and is computed as (11):

$$C \odot (d_1 - d_0) + d_0, \tag{11}$$

where $\odot$ denotes the *external product*, homomorphic to the external $\mathbb{R}$-modulo product between the respective plaintexts. The *external product* is formally defined as (12):

$$TGSW \times TLWE \rightarrow TLWE. \tag{12}$$

The external product operation in homomorphic encryption schemes is known to have two computationally expensive functions: the FFT and add-multiply (AddMul). As illustrated in Figure 2, each $BLINDROTATE$ operation, performed with default security parameters, involves 630 $CMUX$ operations, one for each coefficient of the bootstrapping key polynomial. For each $CMUX$ within the external product, the forward FFT is executed six times, followed by the AddMul operation and the inverse FFT. Our experiments indicate that, for a simple homomorphic comparison of two 16-bit numbers, the FFT is performed 309,846 times. Table 1 details the FFT counts for all basic operations in the example code. Following the bootstrapping process, a key-switching function is applied. This operation also presents an opportunity for optimization, which will be discussed later in this section.

Table 1. FFT counts for 16-bit integer comparison

Operation	Key generation	2-input gates	Mux	Comparison function
FFT counts	15120	3780	7554	309846

3.1. Fast Fourier transform

The forward and inverse FFTs share the same underlying architecture, with the sole distinction lying in the applied multiplicative twiddle factors. In this paper, we only discuss the data-path a forward FFT. The core back-end architecture, generated by SGen, utilizes the Cooley-Tukey algorithm iteratively to perform the FFT operation. As illustrated in Figure 5, SGen implements a fully streaming FFT, thereby enhancing resource utilization. A signal of length 2^n is permuted over 2^t cycles in chunks of size 2^k elements, where $n = k + t$. The generated RTL incorporates additional optimizations, including:

- Simplification of read-only memories (ROMs) containing periodic values by replacing them with constants.
- Replacement of single-value ROMs with constants.

– Simplification of trivial arithmetic operations.
– Pairing of multiplexers and ROMs sharing identical values.
These optimizations further improve the area and resource efficiency of the generated hardware design.

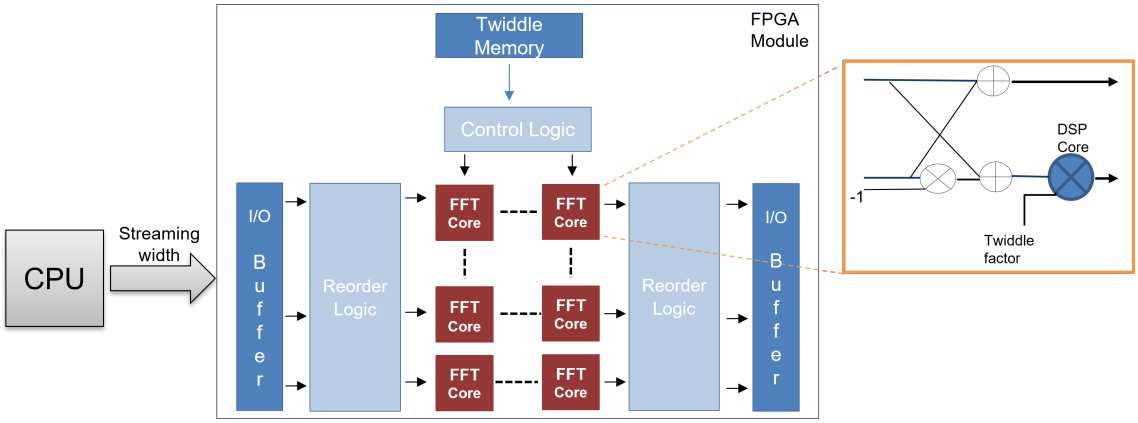

Figure 5. Overview of FPGA implementation of FFT

3.1.1. Design decisions

Automatic RTL generator: SGen offers the flexibility to customize various parameters based on specific design requirements. To evaluate its performance, we compared the latency of SGen and SPIRAL [16] for the FFT of a 64-point, 32-bit fixed-point signal. The results are presented in Table 2. As shown, SGen exhibits superior latency performance compared to SPIRAL. Additionally, SGen provides greater flexibility in selecting data size and type. While SPIRAL is limited to 32-bit fixed-point data, SGen is capable of generating RTL for 64-bit fixed-point data types.

Table 2. Comparison between SPIRAL and SGEN

Radix	2					4			8		
Streaming width	2	4	8	32	64	4	32	64	8	32	64
SPIRAL	165	116	92	57	42	92	40	25	76	46	31
SGEN	131	90	67	34	22	83	26	14	66	30	18

Data type and size: as previously discussed, SGen allows selection from various data types (single, double, and fixed-point). In the TFHE library, FFT operations are performed using the FFTW3 library [17] with double-precision (64-bit) data. We experimented with the single-precision (32-bit) version of FFTW3 for the default TFHE use case, but switching to single-precision floats resulted in inaccurate post-decryption results. Therefore, we opted for a 64-bit word length for the RTL design.

Having determined the word length, we aimed to select the data type (fixed-point or floating-point) by performing simulations. We executed FFTs on the same signals using MATLAB R2020b and the FFTW3 library implementation in Eclipse. The resulting error margins are presented in Table 3. Our goal was to choose precision with the minimal difference compared to the double-precision FFTW3 implementation. These experiments used signals within the operational bounds of TFHE. Research indicated that the integer coefficients of polynomials used for bootstrapping FFTs typically reside in the range $[-64, 63]$. With 64-bit fixed-point representation, the maximum error margin reached e^{-5} (14 decimal points). As expected, increasing the decimal point precision reduced the error margin, ranging from e^{-10} for 30 decimal points to e^{-5} for 14 decimal points. Double-precision floating-point yielded the lowest error margin, reaching e^{-15}. However, for a given transform length, fixed-point implementations are generally faster than floating-point implementations (approximately 3 times faster). Consequently, we opted for 64-bit fixed-point data. It is noteworthy that in fixed-point designs, the precision length for the fractional part does not affect the data path design.

Table 3. Output difference between FFT function on MATLAB and FFTW3 library for different data types

Precision	Double	fixed, 14	fixed, 18	fixed, 22	fixed, 26	fixed, 30
Error margin	e^{-16}	e^{-5}	e^{-6}	e^{-8}	e^{-9}	e^{-10}

Radix size: the Cooley-Tukey algorithm implemented by SGen recursively expresses the FFT as a composite size of $N = N_1N_2$. One of N_1 or N_2 acts as the radix, which controls the number of points processed by the basic computational block of the FFT, often referred to as the butterfly due to its data-path shape (see the magnified section of Figure 5 for a high-level view). The streaming width must be a multiple of the radix; that is, $\log_2(\text{radix})$ should be divisible by $\log_2(N)$. While a higher radix is computationally more complex, it is also more efficient due to the reduced number of multiplications and additions required. To achieve maximum efficiency, high I/O bandwidth between the CPU and FPGA is crucial to maintain a full pipeline at all times.

Streaming width: SGen allows for varying the streaming width from 2 to 256 complex words per cycle, depending on the chosen radix. A fully streaming architecture enables continuous data flow into and out of the system. The Cooley-Tukey FFT data path comprises $O(\log_2 N)$ cascaded stages for a signal of length N. A streaming width of w words per cycle requires $\frac{N}{w}$ cycles to receive the entire input (or output). The next section presents an evaluation of performance (resource utilization and speed) as influenced by both streaming width and radix.

3.1.2. Proposed convolution approach

We propose a new method for performing multiplication using FFT, as shown in Algorithm 2. This method is applicable to signals of length N and utilizes an FFT data path of length $N/2$. We extend the concept of NWC from finite fields to the field of complex numbers. A similar approach for real-valued numbers was previously proposed in [18]. We assume that the twiddle factors are pre-computed. The inputs are real-valued signals of length N, which are folded to create a complex signal of length $N/2$.

To verify the correctness of our approach, we performed polynomial multiplication using the implementation provided by the TFHE library (described in Algorithm 1) and the proposed approach (described in Algorithm 2) on five different input signals. We then calculated the average difference between the corresponding coefficients of the final product signals. We fixed the range of the integer input coefficients to [-63, 64], consistent with the TFHE gadget decomposition function. Our findings indicate that the proposed approach achieves high accuracy even when using a half-size FFT data path. The total error margins were averaged at 7.81253×10^{-9}, with a median of 1.95309×10^{-9}.

3.2. AddMul

The AddMul function performs multiplication followed by successive addition within a loop, as detailed in Algorithm 3. In the TFHE library, this operation is performed in the frequency domain on signals after the FFT operation. The data type for this operation is double-precision floating-point. Based on our experiments, the input values are bounded within the range $[-20733.1, 20316.4]$. We implemented the RTL design using MATLAB Simulink R2020b. The implementation employed 64-bit fixed-point data points with an input representation of $(1, 64, 48)$ and an output representation of $(1, 64, 33)$. This fixed-point implementation resulted in a minimal error margin between e^{-7} and e^{-8} on the output compared to the double-precision representation. While increasing the streaming width can enable faster execution by performing operations in parallel, this needs to be carefully balanced against the resulting increase in resource utilization.

ALGORITHM 3: AddMul operation on length N signal

Input: $aa(x), bb(x) \in C^N$ are complex valued polynomials of degree (N), $N = 1024$
Output: $rr(x) \in C^N$
for $j \leftarrow 0$ **to** N **do**
  $rr(j) += aa(j) \times bb(j)$

3.3. Key switching

Key switching is a well-established procedure in the literature [6], [7], allowing the transition between keys with different parameters. Our analysis of this function's profiling revealed that the shift operation, as presented in (3.3.), is the most time-consuming step. This operation, performed on 32-bit integer data in TFHE, exhibits high frequency and low potential for hardware acceleration. Due to its conversion into a single-cycle CPU instruction, the shift operation will not benefit significantly from an RTL implementation. Therefore, the implementation of this function in RTL is omitted in this work.

$$aij = (aibar >> (32 - (j + 1) * basebit))\&mask \tag{13}$$

4. RESULTS AND DISCUSSION

In this section, we present the simulation results for the described RTL implementation of the FFT and AddMul operations generated by SGen. The implementation uses the default parameters for 128-bit security employed in the TFHE library [7]. These results serve as a proof-of-concept for the RTL designs generated by SGen.

We begin by evaluating the achievable performance of the post-synthesis and implemented FFT design with a streaming width of 4. Subsequently, we analyze the impact of increasing the streaming width on resource utilization, latency, and required I/O bandwidth. Finally, we assess the performance of the RTL design for the AddMul operation.

All evaluations were conducted using the 128-bit default security parameters as defined in the TFHE library [7]. The key-switching key ciphertext has a length of 630 bits and a standard deviation of 2^{-15}. Similarly, the bootstrapping key has a length of 1024 bits and a standard deviation of 2^{-25}. Notably, the TFHE library implementation utilizes 32-bit integers and binary keys.

4.1. Fast Fourier transform

This section provides the implementation results as a proof-of-concept of the FFT design. Simulation, synthesis, and implementation have been done with integrated tools of Xilinx Vivado 2019.2 on Virtex-7 xc7vx1140 FPGA. The achieved running frequency after eliminating critical paths over several iterations is 132 MHz.

4.1.1. Performance evaluation

The design parameters chosen for the FFT in this section are: length $n = 1024$, streaming width $w = 4$, and radix butterfly input size $r = 2$. These parameters were selected after evaluating various configurations through simulations, as shown in Table 4. Latency refers to the number of clock cycles after which the output begins streaming.

Table 4. Latency comparison of FFT design module based on streaming width and radix post simulation on Virtex-7 FPGA on Xilinx 2019.2

Streaming idth	4		8		16		32			64			128			256		
Radix	2	4	2	4	2	4	2	4	32	2	4	32	2	4	32	2	4	32
Latency	716	752	401	410	242	236	161	146	170	114	100	116	87	72	86	70	56	68

The streaming width can be configured between 4 and 256, with corresponding available options for radix size being 2, 4, or 32. Streaming width is a crucial factor during architecture design, as the fully parallel RTL design can process w complex inputs per cycle. Table 5 presents the required I/O bandwidth for full utilization of the pipelined architecture.

Table 5. IO bandwidth to fully utilize the architecture of 1024 length FFT module, for 64-bit complex input type

Streaming width	Input bandwidth (GB/s)	Output bandwidth (GB/s)
4	8.4	8.4
8	16.8	16.8
16	33.16	33.16
32	67.32	67.32
64	13.64	13.64
128	27.128	27.128
256	54.256	54.256

For the utilized Virtex-7 FPGA, a streaming width of 4 and radix of 4 were chosen to optimize resource utilization compared to radix 2. Higher bandwidth can be achieved by storing polynomials in on-chip SRAM (potentially generated by other on-chip cores) or in high-bandwidth external memory such as DDR4 or HBM. Table 6 shows the resource utilization for the FFT design post-implementation. The LUT, LUTRAM, FF, and BRAM utilization is less than 5% of the available resources. The bulk of the utilization is DSP and IO with 28% and 93% respectively.

Table 6. Resource utilization post-synthesis and implementation of FFT design module on Virtex-7 FPGA on Xilinx Vivado 2019.2

Resource	LUT	LUTRAM	FF	BRAM	DSP	IO
Available	712000	283200	1424000	1880	3360	1100
Utilized	24687	2267	32661	80	960	1028

4.1.2. Estimation under influence of streaming width

In this section, we discuss the effect of scaling the streaming width and radix for a 1024-length signal on the hardware cost and performance.

The hardware cost for the development of the FFT design is characterized by the number of digital signal processors (DSPs) and block RAMs (BRAMs) utilized. A DSP refers specifically to a Xilinx 7 series DSP48E1, while a BRAM refers to a 36 Kb Block RAM. The resource utilization estimate is based on the corresponding Xilinx IP core generator, and it is important to note that this is a conservative estimate, as potential optimizations specific to the implementation environment have not been applied. As observed in Figure 6, higher radix designs exhibit improved resource utilization due to the reduced number of multiplications and additions required in their FFT implementation.

Latency the speedup in execution time compared to a CPU was measured against an Intel Xeon Platinum 8170 CPU @ 2.1 GHz. The TFHE library was used with its connector to the FFTW3 library [17], a C subroutine library for performing forward and inverse discrete FFT operations. Note that this evaluation focused on a single, low-level FFT operation performed on a signal of length 1024. In practice, such an operation is rarely used in isolation but is rather a component of higher-level operations (e.g., BLINDROTATE) where the speedup is expected to be more significant. Additionally, we anticipate that the performance gains would be more pronounced on a more powerful FPGA. As shown in Figure 6, the hardware FFT achieves a speedup of nearly 30x with a streaming width of 128, with radix-4 offering the best performance.

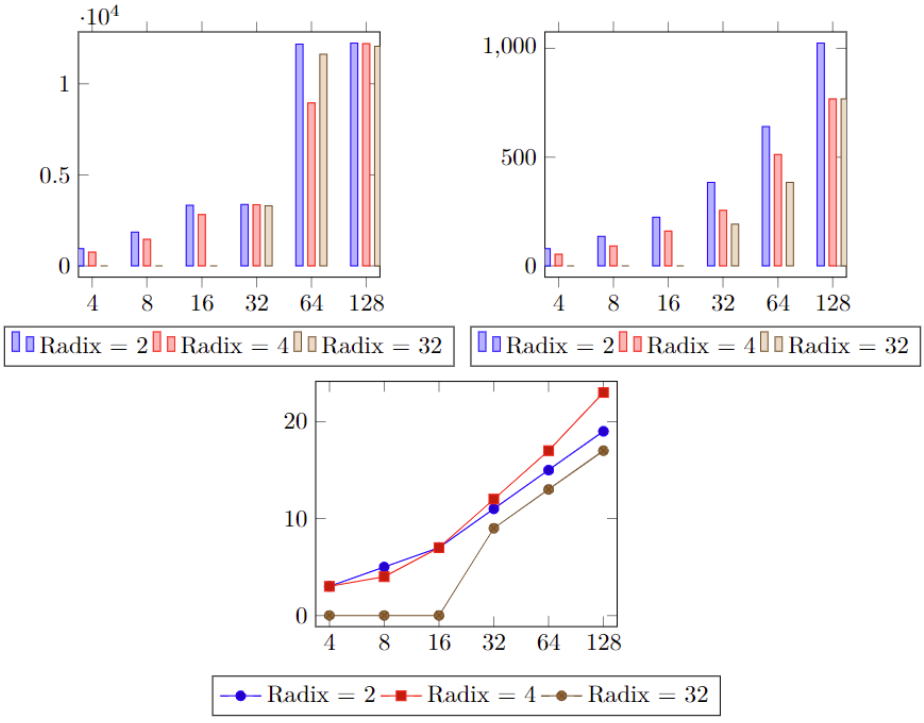

Figure 6. Influence of streaming width of FFT on DSP (left), BRAM (right), and speed-up of a single FFT operation against CPU

4.2. AddMul performance

Complex double-precision number multiplication and addition were performed on the accumulator function in the frequency domain, following the application of Fourier transforms. The function was imple-

mented in RTL as a function of the streaming width to investigate the trade-off between hardware utilization, measured in terms of DSP resources, and latency. The designs met the timing criterion after critical path analysis at a clock frequency of 250 MHz. Timing calculations were based on a signal length of 1024. Speed-up compared to an Intel Xeon Platinum 8170 CPU @ 2.1 GHz was measured. As shown in Table 7, increasing the streaming width has a balancing effect on both DSP utilization and speed-up.

Table 7. Resource utilization and latency post-synthesis of AddMul design module on Virtex-7 FPGA on Xilinx Vivado 2019.2

Streaming width	DSP	Speed-Up
8	288	1.8x
16	576	2.8x
32	1152	3.8x
64	2304	4.5x
128	4608	5x

5. RELATED WORK

With the continued maturation and stabilization of the FHE theory, there has been a growing focus on hardware acceleration to address its performance limitations. This section reviews related literature on hardware acceleration of FHE, both in general and specifically for the TFHE scheme employed in this study. This review provides context for our work within the existing landscape and highlights our key contributions.

5.1. Field-programmable gate array accelerators for Torus fully homomorphic encryption

A recent work by van Beirendonck *et al*. [19] proposes a TFHE accelerator that leverages fixed-point representation for data within the TFHE library. They employ low-precision, ranging from 29 to 31 bits, to represent TFHE data elements. This approach achieves up to 50% reduction in resource utilization compared to their floating-point counterparts. However, it is worth noting that this work may encounter precision limitations as TFHE evolves and incorporates a wider range of parameters.

A recent and highly relevant work by Gener *et al*. [9] presented a vector-based architecture on FPGAs to accelerate TFHE. Their primary focus was accelerating the modular polynomial multiplication in power-of-two cyclotomic rings, a frequent operation in Ring-LWE based FHE schemes. While implementing polynomial multiplication as a vector operation requiring $\mathcal{O}(n^2)$ operations, more efficient convolution methods achieve $\mathcal{O}(n \log n)$ complexity. They utilized the matrix processor (MXP) FPGA [20], programmable to implement up to 16 application-specific instruction set extensions. The MXP was programmed with 10 custom vector instructions specific to TFHE, including vector addition, subtraction, multiplication, and bit extraction. Despite achieving low resource utilization and linear speedup with increasing vector size, their implementation did not match the performance of state-of-the-art CPU or GPU implementations of TFHE. For instance, polynomial multiplication on an ARM Cortex-A9 core was reported to be 2x faster than their MXP implementation. In contrast, our work introduces an efficient polynomial multiplication algorithm based on a modified version of the Fourier transform, as detailed later in this paper.

5.2. Hardware acceleration for non-Torus fully homomorphic encryption schemes

5.2.1. Field-programmable gate array accelerators

Acceleration with designing FPGA based co-processors that work in conjunction with CPU has been widely studied in the literature. These co-processors accelerate one or more operations of homomorphic encryption [21], [22]. The Amazon offers FPGA accelerated cloud for accelerating performance-critical applications [23]. The researchers [24], [25] have designed the encryption/decryption of the Gentry-Halevi (GH) and Brakerski/Fan-Vercauteren (FV) homomorphic schemes respectively on hardware. SIPHER [26] has implemented baseline Homomorphic Encryption prototypes directly in MATLAB using the fixed point toolbox to perform the required integer arithmetic. The implementation being in a high-level language (MATLAB) is easier to adapt and update with improvements [27] present a fast hardware/software co-design implementation of an encryption procedure that leverages the Karatsuba algorithm.

Riazi *et al*. [28] present a hardware architecture for implementing the Cheon-Kim-Kim-Song (CKKS) homomorphic encryption scheme on FPGAs to accelerate Microsoft SEAL. Their design employs number-theoretic transform (NTT) for polynomial multiplication and utilizes several optimized core computation blocks for efficient modular arithmetic. Additionally, they introduce a novel architecture for high-throughput NTT,

achieving a reported speedup of 164-268x compared to the SEAL library [29] running on two Intel FPGAs at 275 MHz and 300 MHz. However, their work has limitations. First, it is limited to small parameter sizes, which restricts its applicability to real-world problems. Second, the CKKS scheme itself does not support bootstrapping, limiting the number of allowed homomorphic multiplications to less than eight. Finally, their multipliers are specifically designed for 27-bit DSP units or multiples thereof, reducing their flexibility for broader use cases.

Another related work that does not employ bootstrapping and consequently results in a shallow circuit depth is presented in [30]. The authors propose an architecture for the well-known FV scheme and implement their design on a Xilinx Zynq UltraScale+ MPSoC ZCU102. They utilize a parallel processing NTT algorithm on top of the single-thread memory-efficient NTT algorithm presented in [31].

Cathebras *et al.* [8] propose another relevant work. They utilize the automatic RTL generator SPIRAL [16] to design a hardware implementation of a fully pipelined residue polynomial multiplier (RPM) for accelerating the full residue number system (RNS) variant of the FV scheme. An important contribution of their work is the decoupling of twiddle factors from the architecture itself, which enhances its versatility and enables it to handle various finite field parameters. The experimentation was performed on an Alpha-Data board ADM-PCIE-7V3, equipped with a Xilinx Virtex-7 xc7vx690t at 200 MHz. For a depth-20 setting, they achieved an estimated speedup for residue polynomial multiplications exceeding 76 during ciphertext multiplication and 16 during re-linearization. Table 8 summarizes the FPGA- and GPU-based implementations of different FHE schemes discussed in this section, highlighting their environments, targeted schemes, and implementation types.

Table 8. Summary of FHE implementations on FPGA/GPU

Implementations	Environment	FHE scheme	Type of implementation
HEAX	Stratix 10 GX 2800	CKKS	RNS+NTT
HEAT	Xilinx Zynq UltraScale+ MPSoC ZCU102	FV	RNS+NTT
Kim	Xilinx UltraScale FPGA	HEAAN	RNS+NTT
Cathebras	Xilinx Virtex 7 xc7vx690t	FV	RNS
F1	Cycle Accurate Simulator	BGV	RNS
HEAWS	Amazon AWS FPGA	FV SHE	RNS

5.2.2. Application-specific integrated circuit accelerators

Several research efforts have focused on designing application-specific integrated circuits (ASICs) to accelerate homomorphic encryption (FHE) computations. These efforts stem from the DARPA data protection in virtual environments (DPRIVE) program, which aims to develop hardware architectures that enable FHE computations within a factor of ten of their unencrypted counterparts.

TREBUCHET [32], [33], F1 [34], CRATERLAKE [35], HERCULES [36], and BASALISC [37] represent notable examples of such hardware design efforts. These frameworks provide comprehensive toolchains for designing FHE accelerator chips, encompassing aspects like architecture exploration, design space optimization, and hardware implementation. While these frameworks share the common goal of accelerating FHE computations, they differ in their underlying architectural choices, targeting different FHE schemes and application domains.

5.2.3. Graphics processing unit accelerators

In addition to FPGA-and ASIC-based acceleration, there has been a significant body of research exploring GPU acceleration for HE. Wang *et al.* [38] investigated the feasibility of using GPUs to accelerate the GH FHE scheme, the first practical instantiation of HE [39]. Their work demonstrated a roughly tenfold performance improvement for critical operations using medium-sized FHE parameters. Similar findings were reported in subsequent works by [40]–[45], who explored the acceleration of other HE scheme variants on GPUs. The TFHE scheme was also targeted for GPU acceleration in cuFHE [46] and nuFHE [47], achieving performance improvements of two orders of magnitude compared to CPU implementations. However, despite the significant speedups offered by GPUs, FPGAs generally provide better performance per watt. This factor was a primary motivator for our choice of FPGAs as the acceleration platform for TFHE.

6. CONCLUSION

In this work, we present hardware implementations on FPGAs for the key building blocks of the bootstrapping procedure in TFHE: the FFT and AddMul operations. We leverage the SGen and Simulink tools for RTL generation and evaluate hardware resource utilization and latency for various configuration parameters. Our evaluations demonstrate that the bootstrapping process can benefit from the efficiency of FPGAs in terms of execution time. Furthermore, we propose a novel algorithm for polynomial multiplication using an extension of the negative convolution technique in the complex domain. This algorithm leverages a length-N data path architecture to perform the FFT of length-$2N$ signals, effectively reducing hardware resource requirements. Our experimental results demonstrate a speedup of up to 30 times for the FFT compared to its CPU implementation.

ACKNOWLEDGEMENTS

This work was supported by the Department of Research and Innovation , Rabdan Academy, through a grant for article processing charges. We appreciate their contribution to the dissemination of our research findings.

REFERENCES

[1] C. Dwork, "Differential privacy," in *International Colloquium on Automata, Languages, and Programming*, Springer, 2006, pp. 1–12, doi: 10.1007/11787006_1.

[2] A. C.-C. Yao, "How to generate and exchange secrets," in *227th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, Toronto, ON, Canada, 1986, pp. 162-167, doi: 10.1109/SFCS.1986.25.

[3] M. Sabt, M. Achemlal and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," *2015 IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, 2015, pp. 57-64, doi: 10.1109/Trustcom.2015.357.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178., doi: 10.1145/1536414.1536440.

[5] S. Halevi and V. Shoup, "Bootstrapping for helib," in *Journal of Cryptology*, vol. 34, no. 1, p. 7, 2021, doi: 10.1007/s00145-020-09368-7.

[6] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2015, pp. 617–640, doi: 10.1007/978-3-662-46800-5_24.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020, doi: 10.1007/s00145-019-09319-x.

[8] J. Cathebras, A. Carbon, P. Milder, R. Sirdey, and N. Ventroux, "Data flow oriented hardware design of rns-based polynomial multiplication for she acceleration," *IACR Transactions on Cryptographic Hard- ware and Embedded Systems*, vol. 2018, no. 3, pp. 69–88, 2018, doi: 10.13154/tches.v2018.i3.69-88.

[9] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, "An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus," in *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*, 2021.

[10] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009, doi: 10.1145/1568318.1568324.

[11] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2010, pp. 1–23, doi: 10.1007/978-3-642-13190-5_1.

[12] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems," *International Symposium on Cyber Security Cryptography and Machine Learning*, pp. 17-28, 2021, doi: 10.1145/3474366.3486924.

[13] R. E. Crandall and C. Pomerance, *Prime numbers: a computational perspective*, Springer, vol. 20052, pp 443–539, 2005, doi: 10.1007/0-387-28979-8_9.

[14] J. Klemsa, "Fast and error-free negacyclic integer convolution using extended fourier transform." *International Symposium on Cyber Security Cryptography and Machine Learning*, vol. 12716, pp. 282-300, Jul. 2021, doi: 10.1007/978-3-030-78086-9_22.

[15] F. Serre and M. Püschel, "A DSL-Based FFT Hardware Generator in Scala," *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Dublin, Ireland, 2018, pp. 315-3157, doi: 10.1109/FPL.2018.00060.

[16] P. Milder, F. Franchetti, J. C. Hoe, and M. Puschel, "Computer generation of hardware for linear digi- tal signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems (TO- DAES)*, vol. 17, no. 2, pp. 1–33, 2012, doi: 10.1145/2159542.2159547.

[17] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," in *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.840301.

[18] R. Matusiak, "Implementing fast fourier transform algorithms of real-valued sequences with the tms320 dsp platform," *Application report-Texas Instruments*, Aug. 2001.

[19] M. van Beirendonck, J.-P. D'Anvers, F. Turan, and I. Verbauwhede, "Fpt: A fixed-point accelerator for torus fully homomorphic encryption," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 741–755, doi: 10.1145/3576915.3623159.

[20] A. Severance and G. G. F. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor," *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Montreal, QC, Canada, 2013, pp. 1-10, doi: 10.1109/CODES-ISSS.2013.6658993.

[21] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of Homomorphic Encryption," *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2014, pp. 1-6, doi: 10.1109/HPEC.2014.7040950.

[22] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor," in *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 193-206, Jun. 2017, doi: 10.1109/TETC.2016.2619669.

[23] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA," in *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185-1196, Aug. 2020, doi: 10.1109/TC.2020.2988765.

[24] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating Fully Homomorphic Encryption in Hardware," in *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509-1521, Jun. 2015, doi: 10.1109/TC.2014.2345388.

[25] A. C. Mert, E. Öztürk, and E. Savaş, "Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353-362, Feb. 2020, doi: 10.1109/TVLSI.2019.2943127.

[26] D. B. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) — FPGA implementation using Simulink," *2012 IEEE Conference on High Performance Extreme Computing*, p. 157, 2012, doi: 10.1109/HPEC.2012.6408672.

[27] V. Migliore *et al*., "A high-speed accelerator for homomorphic encryption using the karatsuba algorithm," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–17, 2017, doi: 10.1145/3126558.

[28] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309, doi: 10.1145/3373376.3378523.

[29] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-seal v2. 1," in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 3–18, doi: 10.1007/978-3-319-70278-0_1.

[30] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Washington, DC, USA, 2019, pp. 387-398, doi: 10.1109/HPCA.2019.00052.

[31] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptopro- cessor," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2014, pp. 371–391, doi: 10.1007/978-3-662-44709-3_21.

[32] D. B. Cousins *et al.*, "Trebuchet: Fully homomorphic encryption accelerator for deep computation," in *arXiv preprint*, 2023, doi: 10.48550/arXiv.2304.05237.

[33] M. Cinque, G. De Tommasi, S. Dubbioso, and D. Ottaviano, "RPUGuard: Real-Time Processing Unit Virtualization for Mixed-Criticality Applications," *2022 18th European Dependable Computing Conference (EDCC)*, Zaragoza, Spain, 2022, pp. 97-104, doi: 10.1109/EDCC57035.2022.00025.

[34] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252, doi: 10.1145/3466752.3480070.

[35] N. Samardzic *et al.*, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187, doi: 10.1145/3470496.3527393.

[36] R. Cammarota, "Intel heracles: Homomorphic encryption revolutionary accelerator with correctness for learning-oriented end-to-end solutions," in *CCSW'22: Proceedings of the 2022 on Cloud Computing Security Workshop*, 2022, p. 3, doi: 10.1145/3560810.3565290.

[37] R. Geelen *et al.*, "Basalisc: programmable hardware accelerator for bgv fully homomorphic encryption," *arXiv preprint*, 2022, doi: 10.48550/arXiv.2205.14017.

[38] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the Feasibility of Fully Homomorphic Encryption," in *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698-706, Mar. 2015, doi: 10.1109/TC.2013.154.

[39] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2011, pp. 129–148, doi: 10.1007/978-3-642-20465-4_9.

[40] W. Dai, Y. Doröz, and B. Sunar, "Accelerating NTRU based homomorphic encryption using GPUs," *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, 2014, pp. 1-6, doi: 10.1109/HPEC.2014.7041001.

[41] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans. Springer,* 2015, pp. 169–186, doi: 10.1007/978-3-319-29172-7_11.

[42] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018, doi: 10.13154/tches.v2018.i2.70-95.

[43] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme," in *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 941-956, 1 April-June 2021, doi: 10.1109/TETC.2019.2902799.

[44] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. K. M. Mi, "Multi-GPU Design and Performance Evaluation of Homomorphic Encryption on GPU Clusters," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 379-391, Feb. 2021, doi: 10.1109/TPDS.2020.3021238.

[45] A. A. Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "PrivFT: Private and Fast Text Classification With Homomorphic Encryption," in *IEEE Access*, vol. 8, pp. 226544-226556, 2020, doi: 10.1109/ACCESS.2020.3045465.

[46] W. Dai and B. Sunar, "Cuda-accelerated fully homomorphic encryption library," 2023.

[47] W. Dai and B. Sunar, "A GPU implementation of fully homomorphic encryption on torus," 2023. [Online]. Available: https://github.com/nucypher/nufhe. (accesed: Jan. 2022, 10).

BIOGRAPHIES OF AUTHORS

Saru Vig received her B.E. degree in electronics engineering, from Manipal University, India, in 2012, the M.Sc. degree in computer engineering from University of Southern California, USA, in 2014, and the Ph.D. degree in computer engineering from Nanyang Technological University, Singapore, in 2020. She is currently a research scientist with the Institute of Infocomm Research, A*STAR, Singapore. Her research interests include but not limited to hardware security, homomorphic encryption, FPGA prototyping, computer architecture, and RTL coding. She can be contacted at email: saruvig03@gmail.com.

Ahmad Al Badawi is an Assistant Professor at the Department of Homeland Security in Rabdan Academy, UAE. He earned his Ph.D. in Electrical and Computer Engineering from the National University of Singapore, following his M.Sc. from the Jordan University of Science and Technology and B.Sc. from Al-Balqa' Applied University. His research spans the intersection of security and computation, encompassing modern cryptography, privacy-preserving computation, parallel processing, combinatorial optimization, and multiprocessor task scheduling. He can be contacted at email: ahmad@u.nus.edu.

Mohd Faizal bin Yusof is an Associate Researcher - Lecturer at Rabdan Academy, UAE. He holds a Bachelor of Science in Electrical Engineering from Northwestern University (1998) and MBA in Technology Entrepreneurship from Universiti Teknologi Malaysia (2008). He is an experienced blockchain researcher, university technology transfer officer, software developer, and former start-ups entrepreneur. He has had the opportunities to hold a number of senior management and technical positions in software technology companies. He has extensive experience in intellectual properties (IP) protection and IP commercialization within university ecosystem. He has hands-on experience in launching cryptocurrency initial coin offering (ICO) and secure token offering (STO). He can be contacted at email: myusof@ra.ac.ae.