ATHENA: Accelerating Torus Fully Homomorphic Encryption on Energy-Efficient Heterogeneous Architecture

Yanze Wu
Cyber Security Engineering Department
George Mason University
Fairfax, VA, USA
ywu42@gmu.edu

Md Tanvir Arafin
Cyber Security Engineering Department
George Mason University
Fairfax, VA, USA
marafin@gmu.edu

Authors Copy

***Abstract*—Fully Homomorphic Encryption (FHE) enables privacy-preserving computations on encrypted data with strong security guarantees. Torus-based FHE (TFHE) emerges as a promising candidate among FHE variants due to its efficient Boolean logic operation and unlimited computational depth. However, it heavily relies on bootstrapping, a computationally intensive technique. Although there has been significant progress in improving the throughput and latency of the bootstrapping process, there exists a gap in the energy efficiency research of this process without compromising its speed. Also, energy-efficient implementation of TFHE is a key requirement for its application in energy-constrained systems.**

This work introduces ATHENA, an energy-efficient bootstrapping accelerator for TFHE built on a heterogeneous Versal adaptive system on chip (ASoC) platform to address this gap. ATHENA partitions the bootstrapping workload into different parts of ASoC: the serial operations are handled by the processing system (PS), the compute-intensive torus multiplications are mapped to the adaptive intelligent engine (AIE), and the memory and communication operations are allocated on the programming logic (PL). ATHENA derives a wavefront array-based energy-efficient multiplier, achieving a higher ($2\times$) improvement in throughput over a similar implementation (Saber-NTT, TCAS '23). ATHENA uses this multiplier to deliver an end-to-end bootstrapping accelerator on the Versal VCK-190 platform. ATHENA delivers $7\times$ better energy efficiency for bootstrapping than GPU-based CuFHE (RTX 3090) and outperforms existing complete FPGA designs, such as YKP (HPEC '22) by demonstrating 17.5%, and 35.6% decrease in latency and energy consumption. To the best of our knowledge, this is the first *PS+PL+AIE*-based heterogeneous TFHE accelerator on Versal ASoCs. ATHENA's code and experimental artifacts are published at https://github.com/SPIRE-GMU/tfhe-aie/ for evaluation and reproducible research.

***Index Terms*—Torus Fully Homomorphic Encryption (TFHE), Programmable Bootstrapping, Scalable Polynomial Multiplier, Coarse-grained reconfigurable architecture (CGRA), Heterogeneous Architecture.**

I. Introduction

From personalized healthcare to autonomous driving, modern advancements in machine learning and big data algorithms increasingly require access to users' sensitive private data. Though technologically innovative, this data-for-intelligence paradigm raises significant concerns about privacy risks. Fully Homomorphic Encryption (FHE) is a promising solution to mitigate such information leakage by enabling direct computations on encrypted data.

However, computational complexity limits the practical application of FHE. For example, the first FHE algorithm [1], proposed in 2011, took up to 30 minutes to calculate an encrypted AND gate on a generic CPU. Since then, several FHE schemes have been developed to speed up the process. A recent scheme, Torus FHE (TFHE), notably reduced the encrypted AND gate computation to 13 ms on a CPU. Despite such significant advances, processing with TFHE is still around $10^8\times$ slower than its unencrypted counterpart.

The most computationally expensive operation in the TFHE is bootstrapping, a mandatory process for maintaining ciphertext integrity over extended operations. Hence, there has been significant interest in accelerating this critical operation. Graphical processing units (GPUs) are often preferred [2], [3] due to their parallel processing capabilities. Field-programmable gate arrays (FPGAs) are also actively explored to accelerate TFHE workloads [4]–[6].

Interestingly, while most existing work on TFHE acceleration prioritizes speed metrics, other critical factors like power consumption and energy efficiency are often overlooked. Achieving improvement in speed by scaling hardware resources is straightforward, but it also comes at the cost of higher power demands. Therefore, additional research on hardware-software-based co-design and optimization is required to realize TFHE-based computation on power-constrained devices.

Fortunately, due to the computation demand for artificial intelligence (AI) workloads, significant development has been made in recent years in the energy-efficient hardware design. For example, specialized accelerators such as Google's Edge TPU [7], NVIDIA's Jetson platform [8], Tenstorrent's GraySkull [9], and Apple's Neural Engine [10] are designed to perform complex AI inference tasks with minimal power consumption. Some of these AI co-processors (such as Tenstorrent's GraySkull and AMD-Xilinx's adaptive intelligent engines (AI Engines) [11]) leverage a spatial array coarse-grained reconfigurable (CGRA) architecture where a large

Funding agencies and acknowledgment details will be provided here in the final version of the manuscript.

number of processing elements (PEs) are arranged in a multi-dimensional grid connected via a fast network-on-chip to parallelize a complex task. Although these architectures are built with AI workload in mind, their application in cryptography has the potential to usher in a new era of secure computation at the edge.

Hence, this work proposes an energy-efficient TFHE accelerator realized on AMD-Xilinx's Versal adaptive system-on-chip (ASoc) platform. The main contributions of this work are given below.

C1. We investigate dataflow-based spatial accelerator designs for torus polynomial multiplication and derive a *novel* high-throughput FFT-based polynomial multiplier for the Versal ASoC. This design leverages the Chinese Remainder Theorem (CRT) and Garner's Algorithm to realize high-precision FFTs using low-precision floating-point units present in the PEs of the AI Engine.

C2. We further improve polynomial multiplication by understanding the bottlenecks in the FFT implementation and deriving a wavefront architecture to accelerate the direct polynomial multiplication. This second multiplier significantly decreases the latency of polynomial multiplication and improves the throughput. For traditional TFHE parameters, this multiplier has a throughput of up to 44 Gbps.

C3. Utilizing the low-latency wavefront polynomial multiplier, we develop an end-to-end bootstrapping accelerator, ATHENA, on the Versal ASoC platform. The overall design strategy presents a unified dataflow-based approach for accelerating complex workloads on Versal ASoCs.

C4. We compare ATHENA with existing bootstrapping accelerators in the literature. ATHENA delivers 7× better energy efficiency for bootstrapping than GPU-based CuFHE (RTX 3090) and outperforms existing complete FPGA designs, such as YKP (HPEC '22) by demonstrating 17.5%, and 35.6% decrease in latency and energy consumption.

C5. We also open-source our codes at **https://github.com/SPIRE-GMU/tfhe-aie/** for artifact evaluation and reproducible research.

The rest of the paper is organized as follows. The basics of TFHE and CGRA primitives are given in the next section. Section III presents the details of developing a polynomial multiplier on CGRA-based architectures. Section IV provides the benchmarking results and performance comparison among the proposed accelerator and the existing work in the literature. Section V concludes the paper.

II. Preliminaries

This section overviews TFHE and its core operation: programmable bootstrapping. Detailed information for TFHE can be found in [12].

A. Notation

This work uses capital letters to denote polynomials and lowercase letters for their coefficients. $\mathbb{Z}$ is the set of integers, $\mathbb{R}$ is the set of real numbers, and $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ represents the torus whose elements are real numbers modulo 1 lying in the interval [0,1). $\lfloor \cdot \rceil$ is a rounding to the nearest integer. μ and σ indicate the mean and standard deviation of the Gaussian distribution.

B. Torus Fully Homomorphic Encryption

TFHE is a fully homomorphic encryption scheme whose security is based on the hardness of lattice problems. TFHE uses the torus to represent the coefficients of the encrypted ciphertext, which differs from the bit-based representation found in other FHE schemes. TFHE computations are based on torus polynomial rings of the form $\mathbb{T}[X]/(X^N+1)$, where each element of this ring is an $N-1$ degree polynomial as shown in Equation 1.

$$a(X) = a_0 + a_1X^1 + ... + a_{N-1}X^{N-1} \text{ where } a_i \in \mathbb{T} \quad (1)$$

In practical libraries, such as [12], a_i is encoded using 32 or 64-bit integers. We will follow the same approach and represent torus polynomials using the integer ring $\mathcal{R}$ for internal computations. Note that, for $\mathcal{R}_q$, modular reductions will be centered around zero, meaning that for a 32-bit modulus q, coefficients are constrained in the range $[-2^{31}, 2^{31}-1]$.

Encryption: Assume a message M $\in \mathcal{R}_q$, two positive integers p and q, where $p \leq q$ and both of them are power of two, and $\Delta = q/p$. To encrypt this message, TFHE first generates a secret key $S = (S_0, S_1, ..., S_{k-1}) \in \mathcal{R}^k$ consisting of k-random polynomials sampled from uniform binary distributions. It also generates a uniformly random mask $A = (A_0, A_1, ..., A_{k-1}) \in \mathcal{R}_q^k$. Then, the ciphertext C is defined as the tuple $(A_0, A_1, ..., A_{k-1}, B)$, where, $B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E$. Here, the polynomial E is considered as noise, whose coefficient is sampled from a Gaussian distribution. This ciphertext is defined as the general learning with error ciphertext (GLWE).

Decryption: To decrypt a GLWE ciphertext $(A_0, A_1, ..., A_{k-1}, B)$, with the knowledge of the secret key S one can reduce the body B by performing $B - \sum_{i=0}^{k-1} A_i \cdot S_i = \Delta M + E$. Then, the message M is recovered with a rounding operation $M = \lfloor (\Delta \cdot M + E)/\Delta \rceil$.

Homomorphic Operation: To perform universal homomorphic operations, TFHE requires solving both the *addition* and *multiplication* problems. Addition can be directly performed on GLWE ciphertexts, as they are homomorphically additive. On the other hand, to solve the homomorphic multiplication (*i.e.,* for messages M_1 and M_2 calculate $GLWE(M_1 \cdot M_2)$), TFHE introduces two more types of ciphertext: GLev (Generalized Level) and GGSW (General Gentry-Sahai-Waters).

GLev is a vector form of GLWE ciphertext, calculated by scaling the same message M with factors $\frac{q}{\beta^j}$ *i.e.,*

$$B^j = \sum_{i=0}^{k-1} A_i^j \cdot S_i + \frac{q}{\beta^j} M + E^j \quad (2)$$

where β, generally power of two, is called the base, l is the number of levels, and $j \in \{1, 2..., l\}$, as shown in Fig. 1.

Authors Copy

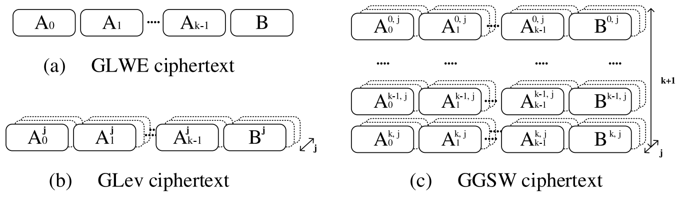


Fig. 1. Three types of ciphertext used in TFHE. (a) GLWE is the basic ciphertext format, consisting of a vector of polynomials. (b) GLev is a vector of GLWE ciphertexts created by scaling the message M with j different factors. (c) GGSW is a vector of GLev ciphertext designed by multiplying rescaled message M with one of the polynomials of secret key $-S_i$.

The GGSW is made up of a list of GLev ciphertexts by multiplying the scaled message M with one of the polynomials of secret keys, *i.e.*,

$$B^{t,j} = \sum_{i=0}^{k-1} A_i^{t,j} \cdot S_i + \frac{q}{\beta^j}(-S_t)M + E^{t,j} \quad (3)$$

where $t \in \{0, 1, ..., k\}$.

Then, for calculating $GLWE(M_1 \cdot M_2)$, we use the ciphertexts $GGSW(M_1)$ and $GLWE(M_2)$ and compute their *external product* (denoted with $\boxdot$):

$$GLWE(M_1 \cdot M_2) = GGSW(M_1) \boxdot GLWE(M_2) \quad (4)$$

The external product computation requires decomposing $GGSW(M_1)$ into a list of $GLev(M_1^t)$,where $t \in \{0, 1, ..., k\}$. $GLWE(M_2)$ is also decomposed into $GLev(M_2)$. Then, we perform element-wise polynomial multiplication and accumulate the results to calculate $GLWE(M_1 \cdot M_2)$ [13]. This overall operation involves $(k + 1) \times l \times (k + 1)$ times polynomial multiplications.

C. Bootstrapping in TFHE

Unfortunately, every homomorphic operation adds noise to the ciphertext, and thus, computing an arbitrary depth function to realize **fully** homomorphic encryption, one needs to manage the noise. Gentry *et al.* [1] solved this problem by introducing bootstrapping. Bootstrapping sustainably manages noise by homomorphically evaluating the decryption of a ciphertext and re-encrypting the result as a fresh ciphertext with low noise [1]. Bootstrapping is a core computation for practically realizable FHE, and **the efficiency of an FHE scheme depends on efficient evaluation of bootstrapping**.

TFHE supports *programmable bootstrapping* that enables the evaluation of custom functions $f(x)$ on encrypted data during bootstrapping. The pseudocode for bootstrapping is presented in Algorithm 1. Below we provide an overview of the process.

Assume a message $m \in \mathbb{Z}_p$ encrypted using an n-bit secret key $s = (s_0, ..., s_{n-1})$, and a mask $a = (a_0, ..., a_{n-1})$, resulting an LWE ciphertext (a, b) . First, we encrypt each bit of s as a GGSW ciphertext, which is called the bootstrapping key $BK = (BK_0, BK_1, ..., BK_{n-1})$. Let V be a polynomial that encodes a function f, and is embedded into a polynomial ring as a GLWE ciphertext. We initialize an accumulator polynomial ACC by rotating V by $-b$ positions. Then, we iteratively update the ACC as

$$ACC \leftarrow (ACC \cdot X^{a_i} - ACC) \boxdot BK_i + ACC \quad (5)$$

This is called blind rotation as s_i is hidden in BK_i under a GGSW. After n rounds of this process, the final result holds the value $f(m)$ encrypted, and the updated LWE ciphertext can be extracted from the first coefficient of ACC. For the details on programmable bootstrapping and the correctness of this process, we refer the reader to [13].

Algorithm 1 TFHE Programmable Bootstrapping

```
0: procedure TFHE_BOOTSTRAP($C_{out}$, BK, V, $C_{in}$)
     input: $c_{in}$ = $(a_0,a_1,...,a_{N-1}, b) \subseteq \mathcal{R}_q$   ▷ LWE ciphertext
     input: BK = $(BK_0,BK_1,...,BK_n - 1) \subseteq \mathcal{R}_q^{(k+1)*(l+1)*(k+1)*n}$   ▷ Bootstrapping key
     input: V $\subseteq \mathcal{R}_q^{k+1}$   ▷ GLWE polynomial
     output: $c_{out} \subseteq \mathcal{R}_q$
1:   ACC ← $V \cdot X^{-b}$   ▷ Initialize ACC
2:   for i =0; i<n; i++ do
3:     ACC ← External_Product(ACC,BK$_i$) + $ACC$;
4:   end for
5:   temp ← LWE_Extract(ACC)
6:   $c_{out}$ ← Key_Switch (temp)   ▷ Switch secret key from GGSW to LWE
```

The *blind rotation procedure* involves n-round external products leading to a large number of polynomial multiplications (*e.g.*, 6000 polynomial multiplications for parameter Set II listed in Table I). Thus, ***blind rotation becomes a computational bottleneck for realizing TFHE on energy-constrained systems.*** Hence, building a scalable and efficient polynomial multiplier for the blind rotation is key to accelerating the bootstrapping process.

This work uses two sets of standard parameters of TFHE implementations for experimental and evaluation purposes as listed in Table I. Set I is found in the CONCRETE Boolean library [14], and Set II is adopted from the common-benchmark TFHE library [13].

TABLE I
THE PARAMETER SETS USED IN THIS WORK.

Parameter Set		I [14]	II [13]
Security	λ	128-bit	110-bit
TLWE dimension	n	586	500
TGLWE dimension	k	2	1
Polynomial size	N	512	1024
Decomposition Base	β	8	10
Decomposition Level	l	2	2

D. Versal Adaptive System on Chip (ASoC)

Field-Programmable Gate Arrays (FPGA) usually consist of a real-time processing unit (PS) and a large amount of programmable hardware logic (PL). Interestingly, the latest generation of FPGA devices from AMD-Xilinx integrates spatial processor elements for accelerating machine learning

Authors Copy

and digital signal processing workloads. For example, the Versal VCK-190 devices contain a grid consisting of 400 PEs (dubbed the adaptive intelligent engine or AI Engine (AIE)) as shown in Fig. 2. Such designs incorporate traditional coarse-grained reconfigurable arrays into the FPGA platform. These hardened PEs in Versal devices enable *fast* implementation of *reconfigurable kernels* for *energy-efficient* applications. Thus, we are entering the era of *heterogeneous FPGA systems* where PS+PL+AIE-based designs will dominate the performance and efficiency of the emerging algorithms on FPGAs.

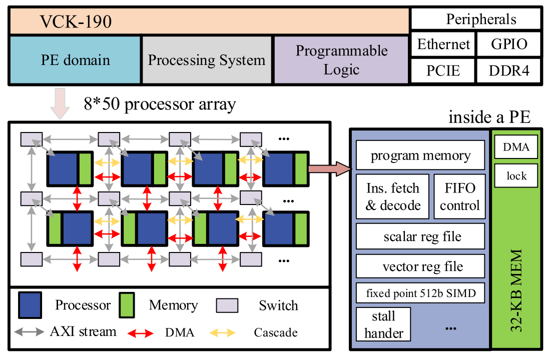


Fig. 2. Architecture of the Versal adaptive system-on-chip (ASoC) platform. It consists of (1) a processing system that executes complex real-time applications, (2) programmable logic to support FPGA-based hardware acceleration, and (3) a PE domain for spatial acceleration tasks.

In the AIE domain, 400 processors are arranged in a two-dimensional grid. Each PE has 32 KB local data memory, 16 KB programmable memory, and a 7-way very long instruction word (VLIW) that supports two read (memory to register), two move (update registers), one write (register to memory), one scalar operation, and one vector operation. A PE has direct access to the neighboring PE's memory in four directions. In addition, each PE has two input and two output AXI-streams of 32-bit width, enabling communication between non-neighboring PEs. A higher bit-width channel crossing the PEs, which is called a cascade, allows a 384-bit stream in-out. Last but not least, a lock mechanism in each PE avoids memory access collision during communication.

III. Implementation of ATHENA

This section demonstrates how to construct the bootstrapping accelerators, ATHENA, on the heterogeneous Versal ASoCs. As discussed in Section II, the efficiency of the accelerator will be dominated by efficient polynomial multiplication used in blind rotation. Hence, first, we focus on building an efficient and scalable polynomial multiplier.

A. Mapping Polynomial Multiplication to Array Processors

We explore Fast Fourier Transform and wavefront-based multiplier architectures to map the polynomial multiplication in the AIE. Our mapping approach is built on the following steps: (1) describing the algorithm with a data Dependency Graph (DG); (2) projecting the DG to a Signal Flow Graph (SFG); (3) mapping the SFG to processor elements.

1) FFT-based Multiplier Design: Direct torus polynomial multiplication has a complexity $\mathcal{O}(N^2)$. Number Theoretic Transformation (NTT) is a well-known solution for reducing this complexity to $\mathcal{O}(N \cdot log(N))$. One can also use a higher precision Fast Fourier Transform (FFT) to replace the NTT operations. For example, in modern CPU-based TFHE libraries such as TEHE-rs [12], polynomial multiplication is realized via the optimized 64-bit FFT libraries such as the FFTW [15].

Generally, FFT is considered faster than NTT on most FPGAs since FPGAs equip hardware-accelerated floating point units (FPU) capable of fast FFT computation. However, the FPUs in FPGAs usually do not support high-precision multipliers, which is required for accurate TFHE operations. As a result, several recent works, *e.g.*, Ye's [4] and Kong's [6], choose NTT to ensure the correctness of the results, while [5] chooses a fixed-point accelerator instead.

The PEs in the Versal AIE contain energy-efficient, low-precision (32-bit) FPUs that are inadequate to support the 64-bit precision required for the standard TFHE operation. To solve this, *we leverage the Chinese Remainder Theorem (CRT) and Garner's algorithm to decompose the high-precision FFT computation problem into several low-precision FFT computations* [16] as shown in Algorithm 2. In the practical TFHE case, where maximum coefficients are bounded from -2^{31} to $2^{31}-1$, this translates to choosing three primes p_1, p_2, and p_3 such that, $p_1 \cdot p_2 \cdot p_3 > 2^{2B}$ $(B = 31)$, and perform the FFT operations on fields with moduli p_1, p_2, p_3 as shown in Fig. 3.

Algorithm 2 High-Precision Polynomial Multiplication

```
 0: procedure POLYNOMIAL MULTIPLICATION($A_{in}$, $B_{in}$, $C_{out}$)
        input: $A_{in}$,$B_{in}$                          ▷ Input polynomials
        output: $C_{out}$
 1: initialize $p_1, p_2, \dots p_\xi$
 2: for i =1; i<= $\xi$; i++ do
 3:     $A_i \leftarrow A_{in}$ mod $p_i$                 ▷ Decompose with moduli $p_i$
 4:     $B_i \leftarrow B_{in}$ mod $p_i$
 5:     $A_i \leftarrow$ FFT($A_i$)                       ▷ low-precision FFT
 6:     $B_i \leftarrow$ FFT($B_i$)
 7:     $Y_i \leftarrow <A_i, B_i>$                       ▷ Elementwise multiplication
 8:     $Y_i \leftarrow$ IFFT($Y_i$)
 9: end for
10: $C_{out} \leftarrow$ Garner($Y_1, \dots, Y_\xi$)      ▷ Recover exact result
```

The dependency graph (DG) for an FFT operation can be represented with the butterfly structure in Fig. 3. This algorithm processes 2N samples across $log_2(2N)$ column-wise stages, creating an inherently sequential workflow. For example, stage-2 requires completion of stage-1's first four samples, stage-3 waits for stage-2's eight samples, *etc*. Although this dependency does not create concerns for a single-processor system, it can negatively amplify latency in a multiple-processor architecture due to cascade stalls.

Next, we derive the signal flow graph from the DG by vertically grouping the computation tasks as shown in Fig. 4. This vertical projection (Fig. 4 (a)) groups tasks by stage,

Authors Copy

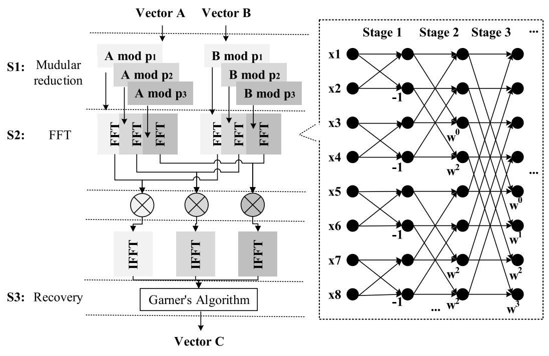
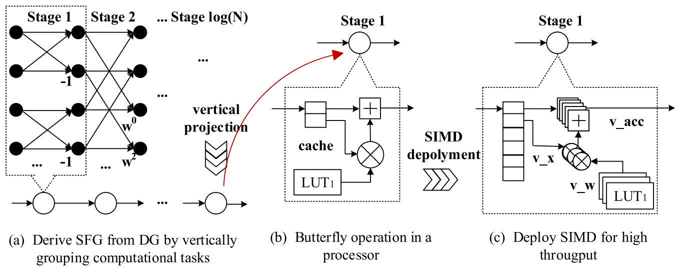


Fig. 3. Steps to perform high-precision FFT with single-precision floating-point numbers.

assigning each processor a fixed butterfly operation.



Stage 1
Stage 2
... Stage log(N)
vertical projection
cache
LUT1
SIMD deployment
v_acc
v_x
v_w
(a) Derive SFG from DG by vertically grouping computational tasks
(b) Butterfly operation in a processor
(c) Deploy SIMD for high througput

Fig. 4. Signal flow graph for the FFT algorithm is derived by (a) vertically projecting computing tasks into a processor. (b) Each processor involves butterfly computation of data samples with twiddle factors. (c) Further acceleration on each PE using the SIMD paradigm.

Finally, we map the SFG into PEs as shown in Fig. 5. Each PE has a dedicated kernel matching its butterfly size, *e.g.*, stage-1: 2-point, stage-2: 4-point, *etc.*. All kernels process 2N samples but vary in execution time. Due to the data dependency mentioned above, PEs in a later stage stall until the preceding PE generates the required results. To reduce this latency, the proposed multiplier works in a pipelined fashion, enabling a continuous stream of FFT tasks.

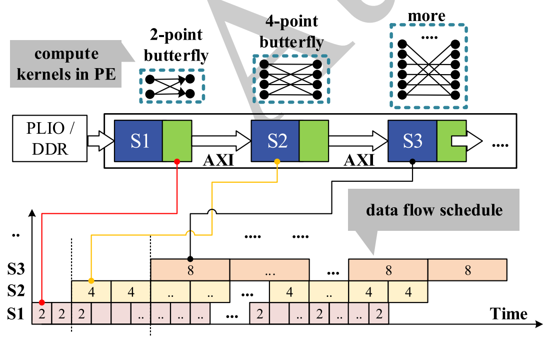


Fig. 5. Processor array and schedule of the workflow. Assuming each PE process is a variable-scale butterfly with specific kernels, the schedule of PE on processing data flow can be derived by considering their execution times. For example, if processor (S2) computes a 4-point butterfly, its execution time will be double that of S1, which computes a 2-point butterfly.

2) Wavefront Polynomial Multiplier Design: Although the FFT-based multiplier outperforms the current FPGA-based designs in terms of throughput, the serial dependency of the butterfly stages results in high latency. To improve both the latency and throughput measures, we explored the implementation of direct polynomial multiplication on AIE and its acceleration.

To simplify the multiplication process, let's first abstract polynomials as vectors, *i.e.*, $A^{1\times N} = [a_0, a_1, ..., a_{N-1}]$ where coefficients $a_i, i \in \{0, ..., N-1\}$ are 32-bit integers (as found in standard libraries such as [12]). When computing a polynomial multiplication $C^{1\times N} = A^{1\times N} \cdot B^{1\times N}$, the coefficients of C are obtained as

$$\begin{aligned} C^{1\times N} &= [c_0, c_1, ..., c_{N-1}] \\ c_0 &= a_0 b_0 - a_1 b_{N-1} - ... - a_{N-1} b_1, \\ c_1 &= a_0 b_1 + a_1 b_0 - a_2 b_{N-1} - ... - a_{N-1} b_2 \\ &... \\ c_{N-1} &= a_0 b_{N-1} + a_1 b_{N-2} + ... + a_{N-1} b_0 \end{aligned} \tag{6}$$

which can be further described with a vector-matrix multiplication as

$$[a_0, a_1, ..., a_{N-1}] \times \begin{bmatrix} b_0 & b_1 & .. & b_{N-1} \\ -b_{N-1} & b_0 & ... & b_{N-2} \\ ... & & ... & \\ -b_1 & -b_2 & ... & b_0 \end{bmatrix} \tag{7}$$
$$= [c_0, c_1, ..., c_{N-1}]$$

Fig. 6 illustrates our derivation of a data dependency graph from Eqn 7. In Fig. 6 (a), the polynomial multiplication is performed using a N inner products between the vector $A^{1\times N}$ and an **extended** vector $B^{1\times 2N}$, which is essentially a convolution process. To do so, $A^{1\times N}$ is horizontally broadcast from the left, while the vectors $B_i^{1\times N}, i \in \{1, 2, ..., N\}$ propagate from the bottom as shown in Fig. 6 (b). This dependency graph decomposes the vector-matrix multiplication into a spatial wavefront-like computation arrangement and highlights the parallelism and data reuse opportunities.

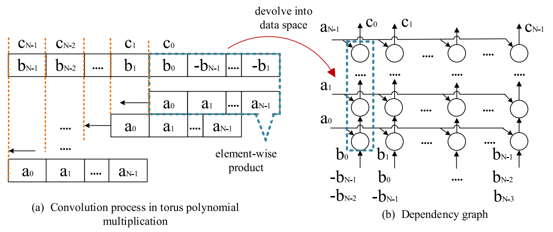
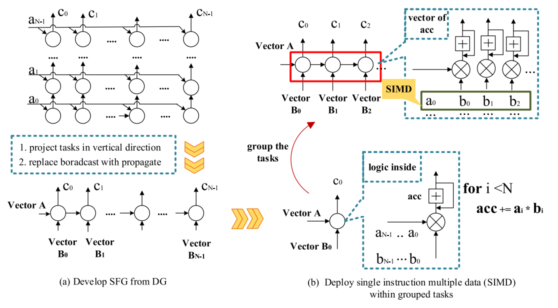


Fig. 6. Converting polynomial multiplication into dependency graph. (a) Multiplication between two vectors can be considered a convolution process. (b) We obtain a data dependency graph by projecting the inner product of vectors into the data space.

Next, we convert this DG into SFG by projecting the column of DG nodes into a single SFG node as shown in Fig. 7 (b). We can perform an additional optimization using the SIMD instructions, essentially using a multiply and accumulate operation on multiple coefficients on a given block cycle, as shown in Eqn. 8 to 10. Details of such an operation can be found in the adaptive computation manual [17]

Authors Copy



1. project tasks in vertical direction
2. replace boradcast with propagate
Vector A
group the tasks
SIMD
vector of acc
logic inside
for i <N
acc += a_i * b_i
(a) Develop SFG from DG
(b) Deploy single instruction multiple data (SIMD) within grouped tasks

Fig. 7. The SFG groups nodes together and deploys them on a single PE using the single instruction multiple data technique. This significantly improves the throughput without increasing the computation complexity.

TABLE II
PERFORMANCE OF N POINT POLYNOMIAL MULTIPLICATION ON THE AI ENGINE

Length N	Freq. (MHz)	Number of PEs	Latency (μs)	Throughput (Gbps)
512	520	8	11.56	2.83
1024	520	16	21.33	3.07
1024	520	32	11.68	5.61
1024	520	256	1.46	44.89
2048	520	32	43.17	3.04

$$\textbf{v8int32}\ v\_a = boradcast < int32, 8 > (\textbf{int32}\ a) \tag{8}$$

$$\textbf{v8int32}\ v\_b = upd\_elem(\textbf{v8int32}\ b, \textbf{int32}\ idx, \textbf{int32}\ b) \tag{9}$$

$$v\_acc = mac(\textbf{v8acc80}\ v\_acc, \textbf{v8int32}\ v\_a, \textbf{v8int32}\ v\_b) \tag{10}$$

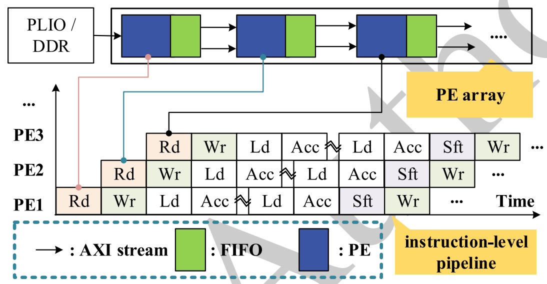


Fig. 8. A polynomial multiplication accelerator based on spatial wavefront layout and its instruction level workflow.

Finally, we map the SFG to processor elements. To reduce the critical path, the PEs that operate on the same data are placed adjacent and aligned with the AXI stream of the 1024-byte depth FIFO. Fig. 8 shows the layout of the wavefront polynomial multiplication accelerator on PEs. This proposed accelerator is scalable to diverse polynomial lengths, as shown in Table II,

B. End-to-end Bootstrapping on Versal ASoC

Fig. 9 illustrates the end-to-end design of ATHENA in Versal ASoC. The bootstrapping computation consists of rotation, decomposition, external product, and composition kernels. These kernels are distributed to independent processors in the AIE domain. Since the FFT-based multiplier shows higher latency, we use the direct multipliers for the overall design. Each multiplier, as introduced in Sec. III-A2, includes 16 high-throughput pipelined PEs.

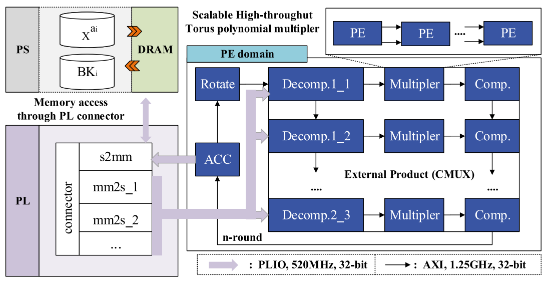


Fig. 9. Overall design of ATHENA on heterogeneous Versal ASoC architecture. The Bootstrapping computation tasks, consisting of rotation, decomposition, composition, and external product, are distributed into PEs connected with a 1.25 GHz AXI stream. The purple arrows indicate the PLIO inout ports of 520 MHz. The external product block consists of 12 multipliers in parallel. Bootstrapping key BK is set up on the PS side.

Blind rotation also involves a large amount of data movement of the bootstrapping key BK, which is initialized outside the accelerator. Hence, a high-bandwidth connection is required between the accelerator and the on-chip memory. The proposed architecture builds the connector with programmable logic (PL), the purple arrows in Fig. 9. The holistic PL-based AXI stream connector has a bandwidth of 149.76 Gbps.

$$\begin{aligned} Freq. \times width \times channel &= 520 \times 32 \times 9 \\ &= 149.76\ (Gbps) \end{aligned} \tag{11}$$

IV. EXPERIMENTAL RESULTS

We benchmark the performance of ATHENA's polynomial multiplication and bootstrapping on Xilinx's VCK-190 platform, consisting of Versal AI core XCVC1902-VSVA2179-2MP-ES chip. Table III lists the configuration of the VCK-190 platform and local workstation used in this work.

TABLE III
CONFIGURATIONS OF VERSAL ASOC AND THE LOCAL WORKSTATION.

Versal ASoC	XCVC1902-VSVA2179-2MP-ES Dual-core ARM Cortex-A72 8 GB DIMM DDR4 , 400 AIEs @1.25GHz
CPU	Model:Intel(R) Core(TM) i9-14900K 32 cores @800-6000MHz (Min-Max) 2.2 MiB L1-I/D, 32 MiB L2, 36MiB L3 2× 32GB DIMM DDR5 @4800MHz Ubuntu 22.04.1 x86_64, GNU bash 5.1.16(1)

A. Accelerating Polynomial Multiplication

Table IV compares FPGA-based accelerators for polynomial multiplication. The multiplication throughput is measured as

$$Throughput = \frac{N \times modulus \times Freq.}{Cycles} = \frac{256 \times log_2(q)}{latency} \tag{12}$$

Authors Copy

TABLE IV
COMPARISON OF $N-$ POINT POLYNOMIAL MULTIPLICATION WITH PRIOR WORKS.

Design	Platform	Parameters (N, $log(q)$)	Freq. (MHz)	Resources. LUTs/FFs/DSP/BRAM/PEs	Latency (μs)	Throughput (Mbps)
Saber-Karatsuba (2021) [18]	UltraScale+	256,13	160	13735/4486/85/6/0	0.52	6415.42
Saber-schoolbook (2022) [19]	UltraScale+	256,13	444	2741/2096/32/0/0	0.15	2.16×10^4
Saber-NTT (2023) [20]	UltraScale+	256,13	416	42440/18660/0/4/0	0.15	1664
Saber-Winograd (2024) [21]	UltraScale+	256,13	556	47654/23872/0/1/0	0.12	2.89×10^4
2D-BFU (2025) [22]	Virtex-7	1024,13	250	7809/6621/24/6/0	3.3	4215.5
2D-BFU (2025) [22]	Virtex-7	1024,13	280	2730/2322/8/3/0	7.5	1854.8
ATHENA-Wavefront	VCK-190	**1024,32**	520	1166/1042/-/-/256	**1.46**	$\mathbf{4.49}\times10^4$
ATHENA-FFT	VCK-190	**2048,32**	520	1158/1040/-/-/34	73.10	$\mathbf{3.30}\times10^4$

TABLE V
COMPARISON OF BOOTSTRAPPING ACCELERATION IN TERMS OF LATENCY, POWER, AND ENERGY CONSUMPTION WITH PRIOR WORK.

Design	Platform	Security λ-bit	Freq. (MHz)	Resource (LUT/FF/DSP/RAM)	Latency (ms)	Power (W)	Energy (mJ)
TFHE library [13]	CPU	128	2900	Intel i7-4910MQ	13.00	76.8	998.4
Concrete [14]	CPU	110	2900	Intel i7-4910MQ	62.00	76.8	4761.6
cuFHE [23]	GPU	110	1700	NVDIA GeForce RTX 3090	9.34	>200	1868
MATCHA [24]	ASIC	110	2000	39.96 mm^2 (16 nm PTM)	>6.8	39.98	271.86
YKP [25]	VU13P	110	180	925K/24K/6240/319Mb	7.53	$\sim$ 50	376.5
YKP [25]	VU13P	80	180	931K/728K/6272/343Mb	19.13	$\sim$ 50	956.5
FPT [5]	Alveo U280	128	200	526K/916K/5494/17.5Mb	0.48	99	47.52
FPT [5]	Alveo U280	110	200	595K/1024K/5980/14.5Mb	0.58	99	57.42
Kong's [26]	VCU128	128	180	480K/715K/2881/48Mb	6.62	/	/
Kong's [26]	VCU128	110	200	414K/625K/1281/40Mb	2.13	/	/
ATHENA-Wavefront	VCK190	128	520	10K/9K/-/-	6.19	39.2	242.65
ATHENA-Wavefront	VCK190	110	520	11K/9K/-/-	10.35	37.6	389.16

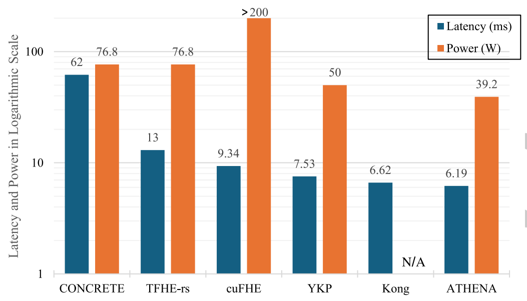


Fig. 10. Logarithmic latency and power consumption of TFHE bootstrapping on various platforms.

A group of accelerators, such as Saber-Karatsuba [18] and Saber-schoolbook [19], streamline the hardware implementation by employing smaller arithmetic bit widths on relatively short polynomials (256-point vector with modulus $log(q) = 13$). These works prioritize speed and throughput, disregarding flexibility and memory utilization. In contrast, **ATHENA's multiplier designs are scalable to polynomial sizes with higher throughput (as much as 2x)** when compared with these designs, as shown in Table IV.

In recent works, another group of multipliers, such as Meta, considers flexibility and builds computing arrays for the FFT/NTT process through a two-dimensional grid of butterfly computing units (BFUs) [22]. However, they suffer from inefficient memory utilization (low bit coefficients versus 32-bit memory width) and restrictive trade-offs between BFU size and polynomial degree. In contrast, the proposed accelerator targets a 32-bit modulus, naturally aligning with the width of standard processor operands. Additionally, the design methodology (DG to SFG to PE) presented in this work develops accelerators from data samples, eliminating the traditional compromise between computing array size and polynomial degree. **The wavefront multiplier design in this work outperforms Meta [26] by 2x in terms of latency and 10x in throughput measurements**.

B. Acceleration of the Bootstrapping Process

The design of accelerators often involves a trade-off between speed and hardware resources. A faster design requires more computational resources, leading to higher power consumption. Therefore, we use latency, power, and energy as the metrics to evaluate design quality. Here, we compare the performance of the proposed TFHE bootstrapping accelerator with recent works, as given in Table V and illustrated in Fig. 10.

In recent designs, GPU-based implementations, such as cuFHE [23], outperform generic CPU implementations, such as CONCRETE [14] and TFHE-rs [13] due to the parallelism opportunities offered by GPUs as shown in Table V. Interestingly, we find that CGRA-based designs, such as ours, can outperform GPUs in terms of energy efficiency and throughput due to the flexible nature of CGRAs. For example, **ATHENA outperforms cuFHE by 7x in energy measures** while keeping the latency on par.

Since Versal ASoC is primarily an FPGA-oriented product, when compared with the FPGA-based TFHE accelerators such as YKP [25] and Kong's [26] design, we find that the introduction of CGRA kernels can improve the energy efficiency of

Authors Copy

cryptographic workload significantly. For example, our benchmark experiments show that **ATHENA outperforms YKP's design by demonstrating 17.5%, and 35.6% decrease in latency and energy consumption** for bootstrapping with a *higher* security parameter.

For the completeness of the work, we also compare ATHENA with *simulated RTL-based designs* and *incomplete* ones, such as MATCHA [24] and FPT [5]. MATCHA is an RTL simulation that shows the promise of ASIC in accelerating the TFHE workload. Interestingly, **ATHENA demonstrates lower latency and energy efficiency than this ASIC implementation** as given in Table V. Such results illustrate the promises of reconfigurable CGRA kernels in efficiently accelerating complex workloads.

On the other hand, FPT [5] builds a TFHE accelerator with extremely low latency and energy consumption. However, as Kong [26] pointed out, *FPT lacks the implementation of the key switching operation*, rendering the overall bootstrapping process incomplete. Moreover, FPT adopts a fixed point number and assumes noise under a theoretical bound of $\sigma^2_{tot} = \sigma^2_{FFT} + \sigma^2_{IFFT} + \sigma^2_{BK}$, but did neither provide experimental noise value introduced in blind rotation, nor the failure probability of decryption [26]. Thus, for a complete design, Kong *et al.* [26] is the front-runner for TFHE accelerator on FPGAs, and ATHENA outperforms Kong's design in terms of latency for $\lambda = 128$. Unfortunately, Kong *et al.* [26] does not provide the design sources or report any power evaluation. Therefore, we could not directly compare their design with ATHENA's in terms of energy efficiency. Overall, Table V provides our detailed benchmarking reports for all discussed platforms and demonstrates the effectiveness of heterogenous (PS+PL+AIE) architecture in TFHE workload acceleration.

V. Conclusion

This work presents a novel, energy-efficient bootstrapping accelerator design, ATHENA, for enabling TFHE operations at energy-efficient heterogeneous edge devices. ATHENA utilizes strong hardware-software co-design techniques by leveraging the reconfigurable processing elements in Versal SoCs. This work should inspire hardware architects, computer designers, and cryptographers to investigate strategies that repurpose and reimagine emerging AI-accelerators and heterogeneous systems to realize complex cryptographic workloads at the edge.

References

[1] 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.

[2] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.

[3] G. S. Cetin, W. Dai, B. Opanchuk, and E. Minibaev, "Cufhe: cuda-accelerated fully homomorphic encryption library," 2018.

[4] T. Ye, R. Kannan, and V. K. Prasanna, "Fpga acceleration of fully homomorphic encryption over the torus," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7.

[5] M. V. Beirendonck, J.-P. D'Anvers, F. Turan, and I. Verbauwhede, "FPT: a fixed-point accelerator for torus fully homomorphic encryption," Cryptology ePrint Archive, Paper 2022/1635, 2022. [Online]. Available: https://eprint.iacr.org/2022/1635

[6] T. Kong and S. Li, "Hardware acceleration and implementation of fully homomorphic encryption over the torus," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 3, pp. 1116–1129, 2024.

[7] S. Hosseininoorbin, S. Layeghy, M. Sarhan, R. Jurdak, and M. Portmann, "Exploring edge tpu for network intrusion detection in iot," *Journal of Parallel and Distributed Computing*, vol. 179, p. 104712, 2023.

[8] H. Halawa, H. A. Abdelhafez, A. Boktor, and M. Ripeanu, "Nvidia jetson platform characterization," in *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28–September 1, 2017, Proceedings 23*. Springer, 2017, pp. 92–105.

[9] J. Vasiljevic, L. Bajic, D. Capalija, S. Sokorac, D. Ignjatovic, L. Bajic, M. Trajkovic, I. Hamer, I. Matosevic, A. Cejkov *et al.*, "Compute substrate for software 2.0," *IEEE micro*, vol. 41, no. 2, pp. 50–55, 2021.

[10] W. Wang, "Technical report: Performance analysis and optimization of mobilenetv2 on apple m2: A detailed study of neural engine and compute unit selection strategies."

[11] AMD, "Versal: The First Adaptive Compute Acceleration Platform," Sep. 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/wp505-versal-acap

[12] Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.

[13] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," Cryptology ePrint Archive, Paper 2018/421, 2018. [Online]. Available: https://eprint.iacr.org/2018/421

[14] 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*, 2020.

[15] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[16] R. J. Fateman, "Exact polynomial multiplication using approximate fft," 2005.

[17] AMD, "Documentation AI Engine Tools and Flows User Guide AMD Adaptive Computation Manual," 2025. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1076-ai-engine-environment/Documentation

[18] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "Lwrpro: An energy-efficient configurable crypto-processor for module-lwr," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1146–1159, 2021.

[19] Y. A. Birgani, S. Timarchi, and A. Khalid, "Area-time-efficient scalable schoolbook polynomial multiplier for lattice-based cryptography," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 12, pp. 5079–5083, 2022.

[20] Y. Cui, Y. Zhang, Z. Ni, S. Yu, C. Wang, and W. Liu, "High-throughput polynomial multiplier for accelerating saber on fpga," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 9, pp. 3584–3588, 2023.

[21] J. Wang, C. Yang, F. Zhang, J. Hou, Y. Meng, S. Xiang, and Y. Su, "A high-throughput and scalable schoolbook polynomial multiplier for accelerating saber on fpga using a novel winograd-based architecture," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 4, pp. 2344–2348, 2023.

[22] Y. Xu, L. Ding, P. He, Z. Lu, and J. Zhang, "Meta: A memory-efficient tri-stage polynomial multiplication accelerator using 2d coupled-bfus," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, no. 2, pp. 647–660, 2025.

[23] W. Dai and B. Sunar, "Cuda-accelerated fully homomorphic encryption library," *Accessed: Jan*, vol. 20, p. 2023, 2019.

[24] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 235–240.

[25] T. Ye, R. Kannan, and V. K. Prasanna, "Fpga acceleration of fully homomorphic encryption over the torus," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7.

[26] T. Kong and S. Li, "Hardware acceleration and implementation of fully homomorphic encryption over the torus," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 3, pp. 1116–1129, 2024.

Author's Copy