IO-Optimized Design-Time Configurable Negacyclic Seven-Step NTT Architecture for FHE Applications

Emre Koçer, Selim Kırbıyık, Tolun Tosun, Ersin Alaybeyoğlu, Erkay Savaş

Faculty of Engineering and Natural Sciences

Sabancı University

Istanbul, Turkey

Abstract—Fully Homomorphic Encryption (FHE) enables computations on encrypted data, making it essential for privacypreserving applications. However, it involves computationally demanding tasks, such as polynomial multiplication, while Number Theoretic Transform (NTT) is the state-of-the-art solution to perform this task. Most FHE schemes operate over the negacyclic ring of polynomials. We introduce a novel formulation of the hierarchical Four-Step NTT approach for the negacyclic ring, eliminating the need for pre- and post-processing steps found in the existing methods. To accelerate NTT operations, the Field-Programmable Gate Array (FPGA)s offer flexible and powerful computing platforms. We propose an FPGA-based, parametric and fully pipelined architecture that implements the improved Seven-Step NTT algorithm (which builds upon the four-step). Our design supports a wide range of parameters, including ring sizes up to 2^{16} and modulus sizes up to 64-bit. We focus on achieving configurable throughput, as constrained by the bandwidth of High-Bandwidth Memory (HBM) bandwidth, and aim to maximize throughput through an IO parametric design on the Alveo U280 FPGA. The implementation results demonstrate a reduction in the area-time-product by $2.08 \times$ and a speed-up of $10.32 \times$ for a ring size of 2^{16} and a 32-bit width compared to the current state-of-the-art designs.

Index Terms—FHE, FPGA, Hardware Acceleration, Four-Step, Seven-Step, Fully-pipelined, NTT, Negacyclic

I. INTRODUCTION

FHE is an advanced cryptographic solution that allows to perform arithmetic operations, including addition and multiplication, directly on ciphertexts without requiring decryption. This capability is particularly critical for applications such as privacy-preserving machine learning. Existing FHE schemes found in the literature are lattice-based [1]–[5], while security of most of these schemes rely on the famous Ring Learning With Errors (R-LWE) [6] problem which is resistant against quantum computing attacks. On the other hand, existing FHE solutions are computationally expensive, making the deployment of those in a real-world application a challenging task.

The core operation in Lattice-Based Cryptography (LBC) is the polynomial multiplication and efficient implementations employ the NTT algorithm to perform it, with a time complexity of $\mathcal{O}(n \log(n))$. Here, n refers to the degree of the polynomial modulus, which is typically between 2^{10} and 2^{16} . The polynomial modulus is generally the cyclotomic polynomial of the form $x^n + 1$ for FHE applications, leading to a *negacyclic* ring. The prime coefficient modulus, which is the other ingredient of the NTT, is generally 32 to 64-bit for implementations that employ the Residue Number System [7].

In this wide range of parameter space, it is challenging to come up with a generic solution that works well for multiple parameter sets. Many solutions operate only with primes in special form [8]-[13].

Recently, there has been a surge of interest among researchers in hierarchical NTT algorithms. These methods are based on the *four-step* approach [14], which treats the input as a 2-D matrix and compute smaller NTTs on both the rows and columns of that matrix. Building on this concept, the sevenstep NTT extends the idea by treating the input as a 4-D hypercube. Hierarchical algorithms have garnered attention of implementers due to their capacity to reduce data dependency and create highly parallelizable NTT architectures [11], [15]-[17]. An algorithmic drawback of existing four-step approach is that it is tailored for the cyclic ring, where the polynomial modulus is $x^n - 1$. Consequently, using it for the negacyclic case necessitates additional steps [11], [17]-[19]. On the other hand, except for [16], existing solutions do not offer high throughput, as they lack fully-pipelined implementations. Given that the NTT is a highly data-intensive algorithm, the primary bottleneck lies in FPGA-to-host communication. Accordingly, throughput-oriented approaches are more feasible compared to those focusing on low latency [20], [21].

To address the aforementioned challenges, our contributions are listed as follows:

- We modify the four-step algorithm to work directly over the negacyclic ring, removing the requirement for both pre-processing and post-processing steps found in existing methods. We show that our solution directly applies to hierarchical NTT approaches in general with any dimensional decomposition of the input polynomial.
- We present an FPGA-based NTT architecture that targets FHE applications, based on the enhanced seven-step algorithm. Our design is optimized for high throughput and I/O efficiency to maximize FPGA-to-host bandwidth utilization. It is configurable at design time, supporting a wide range of parameters, offering flexibility to meet different throughput requirements.
- We provide implementation results for the proposed architecture, targeting the Alveo U280 data center accelerator. Implementation results evidence that the proposed solution significantly outperforms the existing work, by more than two orders of magnitude in terms of average latency for certain scenarios.

II. BACKGROUND

- A. Notation
 - Lowercase italic letters denote integers, such as a. $\Gamma_l(\cdot)$ stands for the bit-reversing function for *l*-bit integers. The logarithm function (log) is base-2.
 - Bold uppercase letters denote matrices, such as A. Similarly, bold lowercase letters are used to denote vectors, such as a. Elements of matrices (vectors) are accessed using the square brackets, such as A[i][j] (a[i]). ⊙ is used to represent element-wise multiplication of vectors or matrices, such as a ⊙ b.
 - *R*_{q,n} denotes the cyclotomic ring of polynomials Z_q[x]/(xⁿ+1). Polynomials are represented by lowercase italic letters, such as a(x). Coefficients of polynomials are represented by the sub-index, such as a_i.

B. Number Theoretic Transform (NTT)

NTT is the state-of-the-art approach for polynomial multiplication. For two polynomials $a(x), b(x) \in \mathcal{R}_{q,n}$, multiplication using the NTT algorithm is performed as follows:

$$\boldsymbol{a}(x) \cdot \boldsymbol{b}(x) = \text{INTT}_n \Big(\text{NTT}_n \big(\boldsymbol{a}(x) \big) \odot \text{NTT}_n \big(\boldsymbol{b}(x) \big) \Big)$$
 (1)

The multiplication in $\mathcal{R}_{q,n}$ is also known as the *negative* wrapped convolution. Similarly, the NTT in $\mathcal{R}_{q,n}$ is referred to as negacyclic NTT, which requires $q = 1 \mod 2n$. Then, a primitive 2n-th root of unity, denoted by ψ , exists in \mathbb{Z}_q , where $\psi^n = -1 \mod q$. For $\hat{\mathbf{a}} = \text{NTT}_n(\mathbf{a}(x))$, the transformation is equivalent to the evaluation $\hat{\mathbf{a}}[i] = \mathbf{a}(\psi^{2i+1})$. Forward and backward NTT can be efficiently implemented using so-called butterfly circuits. Most applications use *Cooley-Tukey* (CT) [22] butterflies for NTT_n and *Gentleman-Sande* (GS) [23] butterflies for INTT_n.

For two coefficients a_i , a_j , CT butterfly is defined as follows:

$$\mathbf{a}'_i = \mathbf{a}_i + \mathbf{a}_j \cdot \zeta \quad \mathbf{a}'_j = \mathbf{a}_i - \mathbf{a}_j \cdot \zeta$$
 (2)

where ζ is called the twiddle factor which is a power of ψ . NTT algorithm utilizing CT or GS butterflies, commonly referred to as the *iterative* form of the NTT [24]. At each iteration, also known as a stage, n/2 butterflies are computed, and there are $\log n$ stages, resulting in $O(n \log n)$ time complexity. Note that a traditional schoolbook multiplication has a time complexity of $O(n^2)$.

1) Four-Step NTT: Bailey's four-step NTT, widely adopted in various studies [11], [15]–[17], [25], is a hierarchical approach that transforms the larger NTT into smaller and independent NTTs. Particularly, the input polynomial with ncoefficients is decomposed into a matrix of size $n_2 \times n_1$. The four steps constituting this algorithm, and which form the basis of its name, are outlined below:

- 1) Perform n_2 independent NTT_{n_1}.
- 2) Transpose the matrix.
- 3) Multiply every element by twiddle factors, element at (i, j) is multiplied by ω^{ij}
- 4) Perform n_1 independent NTT_{n_2} .

As mentioned in Section I, the existing literature on the four-step NTT focused on the cyclic NTT case, where the reduction polynomial is $(x^n - 1)$. In particular, the NTT of $a(x) \in \mathbb{Z}_q[x]/(x^n - 1)$ satisfies $\hat{a}[i] = a(\omega^i)$, where ω is a primitive *n*-th root of unity, with $\omega^{n/2} = -1 \mod q$. To use a cyclic NTT algorithm for performing negative wrapped convolution, such as the above described four-step NTT, additional steps are necessary [18]. To compute $c = a \cdot b$ for $a, b, c \in \mathcal{R}_{q,n}$, the following pre-processing steps must be carried out:

$$\bar{\boldsymbol{a}}_i = \boldsymbol{a}_i \cdot \psi^i, \quad \bar{\boldsymbol{b}}_i = \boldsymbol{b}_i \cdot \psi^i \quad \text{for } 0 \le i < n$$
(3)

Next, \bar{a} and \bar{b} are multiplied according to Equation (1) but using the cyclic NTT routines, with $\omega = \psi^2$. Let \bar{c} represent the result of the multiplication after applying the inverse NTT. Following this, post-processing is done:

$$\boldsymbol{c}_i = \bar{\boldsymbol{c}}_i \cdot \boldsymbol{\psi}^{-i}, \quad \text{for } 0 \le i < n \tag{4}$$

2) Seven-Step NTT: In general, the input of the NTT can be decomposed into any dimensional hyperplane, allowing smaller NTTs to be performed at each dimension. The seven-step NTT is a specific case involving 4-D decomposition, which can be viewed as a one-level recursive application of the previously described four-step approach. Figure 1 provides and outline of the seven-step NTT.

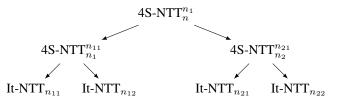


Fig. 1: Illustration of seven-step NTT with recursive four-step NTTs, with 4-D decomposition $n = n_{11} \times n_{12} \times n_{21} \times n_{22}$. 4S-NTT $_n^{n_1}$ denotes four-step NTT where $n = n_1 \times n_2$ while It-NTT denotes the iterative NTT. Twiddle multiplications are not visualized.

C. Memory types and hierarchical memory in FPGA

In contemporary computing, both GPUs and FPGAs are equipped with on-chip and off-chip memories. FPGAs feature an on-chip memory component referred to as Block RAM (BRAM), as well as off-chip memory known as HBM. BRAM enables read/write operations to be completed in a single clock cycle (*cc*), whereas HBM requires multiple *cc* for read/write operations. Specifically for Alveo U280, HBM offers a storage of 8 GB, while the on-chip BRAM can hold up to 41 MB of data. Since FHE applications handle large data volumes, the use of HBM is essential for FPGA-based implementations, though HBM bandwidth can be a bottleneck for data movement. Alveo U280's HBM provides a bandwidth of 8192-bit (4096-bit read and 4096-bit write) at 450 MHz.

III. NEGACYCLIC FOUR-STEP NTT

To address the pre- and post-processing overhead associated with the four-step algorithm described in Section II-B1, we present modified version that directly operates in $\mathcal{R}_{q,n}$. For completeness, Algorithm 1 details the negacyclic four-step NTT. A key aspect of our solution is the formulation of the twiddle factors (Line 7) used during the multiplication loop between two sets of NTTs.

Algorithm 1	Negacyclic	Four-Step	NTT
-------------	------------	-----------	-----

Input: $a(x) \in \mathcal{R}_{q,n}$ **Input:** a primitive 2*n*-th root of unity $\psi \in \mathbb{Z}_q$, $n_1 \cdot n_2 = n$ **Output:** $\hat{\mathbf{a}} \in \mathbb{Z}_q^n$ where $\hat{\mathbf{a}} = \operatorname{NTT}_n(\boldsymbol{a}(x))$ 1: $\mathbf{A} \in \mathbb{Z}_q^{n_1 \times n_2} \leftarrow a \qquad \triangleright$ represe \triangleright represent *a* as a matrix s.t. $\mathbf{A}[i][j] = \boldsymbol{a}_{i \cdot n_2 + j}$ for $i = 0 \to n_2 - 1$ do 2: $\mathbf{A}^{T}[i] = \mathrm{NTT}_{n_1}(\mathbf{A}^{T}[i])$ \triangleright using ψ^{n_2} 3: 4: end for for $i = 0 \to n_1 - 1$ do 5: for $j = 0 \to n_2 - 1$ do 6: $\mathbf{A}[i][j] = \mathbf{A}[i][j] \cdot \psi^{(2 \cdot i - n_1 + 1) \cdot j}$ 7: end for 8: end for 9: for $i = 1 \rightarrow n_1 - 1$ do 10: $\mathbf{A}[i] = \mathrm{NTT}_{n_2}(\mathbf{A}[i])$ \triangleright using ψ^{n_1} 11: 12: end for 13: $\mathbf{\hat{a}} \leftarrow \mathbf{A}$ \triangleright flatten **A** s.t. $\hat{\mathbf{a}}[i \cdot n_2 + j] = \mathbf{A}[i][j]$ 14: return â

IV. PROPOSED IO-OPTIMIZED SEVEN-STEP NTT ARCHITECTURE

This section presents a IO-optimized and pipelined hardware architecture that implements the seven-step NTT algorithm detailed in Section II-B2 incorporating the negacyclic four-step approach outlined in Algorithm 1.

A. Design Principles

1) Row Independence: Recall that the four-step algorithm leverages the independence of rows by processing them separately, meaning there is no dependency between different rows in the matrix representation (see Lines 2-4 and Lines 10-12 in Algorithm 1). This characteristic enables the parallelization of multiple NTT stages. In this work, the advantage of this feature is fully exploited, which is critical for achieving the primary design goal of maximizing throughput.

2) NTT-unrolling: NTT-unrolling involves assigning dedicated BUs to each stage of the iterative NTT, allowing each stage to be processed in every clock cycle, as shown in Figure 2. Ideally, this results in one output per clock cycle with no stalls. However, this approach demands an impractically large number of BUs for large NTT sizes (*e.g.*, $n = 2^{12}$ would require $2^{11} \cdot 12$ BUs, which is unfeasible with current technology). The four-step and seven-step NTTs reduce resource requirements by breaking down the NTT into smaller, more manageable stages, as discussed earlier.

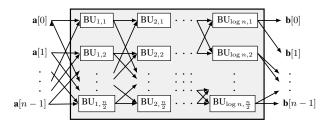


Fig. 2: Loop unrolling, as illustrated for iterative NTT_n , requires $\log n \cdot (n/2)$ Butterfly Units (BUs).

3) Coefficient Throughput: Our solution introduces a unique feature: a parametric hardware generator that operates based on the throughput, denoted by Tp, as well as the seven-step parameters n_{11} , n_{12} , n_{21} , n_{22} for any given $\mathcal{R}_{q,n}$. As previously mentioned, modern FPGAs with HBM face limited bandwidth, which restricts the throughput of the NTT engine. Our approach addresses this issue by considering the I/O bandwidth during the generation of throughput-optimized hardware, aiming to fully utilize the available communication bandwidth.

4) Address Generation: Existing methods in the literature [8], [13], [17], [26]–[29] entail complex address generation units for computing NTTs. In this study, we focus on smaller NTTs compared to previous works, allowing twiddle factors to be efficiently stored in registers.

5) Reduced Twiddle Storage: FHE applications necessitate performing NTT with multiple primes. For instance for $n = 2^{16}$, 60 distinct primes with $\log q = 32$ are used in RNS representation for the R-LWE instance to ensure a promising level of security. Consequently, the set of twiddle factors must be computed for each prime modulus. Our design employs an on-the-fly twiddle generation strategy reducing the cost of the pre-computation.

B. Modular Multiplication

For integer multiplication, we adopt the uneven partitioning technique proposed in [30]. Given that the DSPs in the targeted FPGA, Alveo U280, support 26x17 unsigned multiplication, we partition the multiplication operands into 26-bit and 17-bit parts accordingly. For 64x64-bit multiplication, this approach uses 12 DSPs. For the modular reduction, several approaches have been proposed in the literature. In this work, we adopt Montgomery reduction [31] and its specialized word-level variant, the Word-Level Montgomery (WLM) [24]. This technique is particularly well-suited for efficient hardware implementations.

We employ the WLM as it significantly reduces the number of required multiplications. This approach exploits the fact that the Montgomery factor is -1 for NTT friendly primes which are in the form $q = q_H \cdot 2^{\log 2n} + 1$. For $\log q = 64$ and $n = 2^{16}$, the word size is set to 17, requiring a total of 8 DSP multiplications. Modular Multiplication Units (MMUs) are fully pipelined. The latency of the integer multiplication is 2 *cc*. Similarly, each iteration in the word-level reduction takes 2 *cc*. Consequently, for 64-bit, the latency of MMU is $2 \cdot \left[\frac{64}{\log n+1}\right] + 2$ clock cycles.

C. On-the-fly Twiddle Generation

Recall that the four-step algorithm includes a twiddle multiplication step, where each matrix element is multiplied by a power of ψ , as described in Line 7 of Algorithm 1. It uses n twiddle factors, which are subsequently used. For large values of n, precomputing these factors may become impractical due to the significant memory demands. Consider that FHE applications usually work with multiple values of q, and one needs the set of twiddle factors for all q. Several works can be found in the literature that discusses on-the-fly generation of these twiddle factors. We adapt the strategy discussed in [32] to a fully pipelined design and negacyclic NTT. Let $\mathbf{p}_{i}[i]$ denotes $\psi^{(2 \cdot i - n_{1} + 1) \cdot j}$. Then, $\mathbf{p}_{i+1}[i] = \mathbf{p}_{i}[i] \cdot \mathbf{p}_{1}[i] =$ $\psi^{(2\cdot i-n_1+1)\cdot (j+1)}$. Notice that one can compute \mathbf{p}_j for all j > 2 by only pre-computing \mathbf{p}_1 . This approach reduces storage requirements to only n_1 twiddle factors. For pipelined designs such as ours, the number of pre-computed twiddle factors can be slightly more in practice. In order to produce Tptwiddle factors in each *cc* using the above explained strategy, \mathbf{p}_1 to \mathbf{p}_k are pre-computed where $k = \lfloor (Tp \cdot \mathcal{L})/n_1 \rfloor$ and $\mathcal L$ denotes the latency of the MMU. Then, the On-the-fly Twiddle Generation Unit (TGU) outputs $\mathbf{p}_{i+k}[i']$ in each *cc*, computed based on $\mathbf{p}_j[i']$ in \mathcal{L} clock cycles, for j > k, $Tp \cdot t \leq i' < \min(Tp \cdot (t+1), n_1)$ and some integer $t \leq |n_1/Tp|.$

D. Architecture Overview

In the proposed architecture, there are 11 main modules. These are namely the Iterative NTT Units (INUs) 1-4, Authomorphism (Rotator) Units (AUs) 1-3, Twiddle Multiplication Units (TMUs) 1-3, and TGU. The details of this structure can be found in Figure 3. All the implementation is fully pipelined to maximize the throughput. Generally speaking, INUs perform relatively small NTT operations. TMUs are responsible for multiplying the intermediate coefficients by powers of ψ in between iterative NTTs. AUs are responsible for rotating coefficient outputs for providing correct BRAM read/write addresses for the subsequent iterative NTTs. TGU generates the twiddles used by TMU 2 by implementing the strategy discussed in Section IV-C. Observe that INUs 1-2, along with TMU 1 and AU 1, form a four-step NTT as described in Algorithm 1, referred to as Four-Step NTT Unit (4NU) 1. Similarly, INUs 3-4, TMU 3, and AU 3 constitute another four-step NTT, referred to as 4NU 2. As shown in Figure 1, the seven-step approach is essentially a recursive application of the four-step algorithm.

The proposed seven-step architecture is designed for high throughput. In each cc, the proposed seven-step NTT architecture takes Tp new coefficients as its input and produces Tp coefficients as the result of the corresponding NTT operation. The throughput of all the above-mentioned units matches this value, Tp. Naturally, the output stream becomes valid after a certain number of cc, depending on the architecture's latency.

1) Iterative NTT Units (INUs): INUs 1-4 implement a number of parallel iterative NTTs of size n_{11} , n_{12} , n_{21} and n_{22} , respectively. The number of parallel NTTs are decided by the parameter Tp and the decomposition. For instance, the INU

instantiates Tp/n_{11} iterative NTTs. For the sake of simplicity, we assume Tp is a multiple of all n_{11} , n_{12} , n_{21} , n_{22} . As previously explained, NTT stages are unrolled. Consequently, for INU 1 as en example, there are $(n_{11}/2) \cdot log(n_{11}) \cdot Tp/n_{11}$ BUs. Twiddle factors are also stored in registers. INU 1 does not use BRAMs while INUs 2-4 stores output of the preceding AUs from the pipeline in BRAMs to efficiently and correctly operate. As explained in the previous section, the reason to utilize BRAMs is to perform a set of iterative NTTs on transpose of the matrix compared to the preceding INU. For instance, INU 4 operates on rows while INU 3 operates on columns. As the preceding INU must complete producing all the input coefficients to start the operation. Additionally, newly produced coefficients must be stored in BRAMs. In particular, the size of BRAMs is two times the number of points in the implemented iterative NTT (e.g. $2n_{12}$ for INU 2). At a given time, one half of BRAM blocks are in read mode and the other half of BRAM blocks are in write mode.

2) Twiddle Multiplication Units (TMUs): TMUs implement the twiddle multiplication loop presented in Line 8 of Algorithm 1. TMU 1 and 3 contain a set of Flip-flop (FF)s to store necessary pre-computed twiddle factors. For TMU 1 and 3; n_1 , and n_2 twiddles are stored in FFs, respectively. On the other hand, TMU 2 receives the set of twiddles to multiply from TGU, which reduces its twiddle storage requirement significantly. We would like to note that, the on-the-fly generation is not advantageous for TMU 1 and 3, since the number of pre-computed twiddle factors would be close to the case without the on-the-fly generation (see the computation of k in Section IV-C). To implement the multiplication by twiddles, each TMU contains Tp independently operating and pipelined MMUs to align with the Tp constraint.

3) Authomorphism (Rotator) Units (AUs): Each AU has an internal counter and rotates its input by some pre-defined offset input with respect to the current value of its counter. The bitlength of the counters are distinct for each AU and depends on the parameters decomposition of n, and Tp. Rotation of the coefficients ensure that these coefficients are stored in the desired locations of BRAMs for the subsequent INUs. Thanks to the rotation, the input set of coefficients to be processed by INUs, are stored in independent BRAMs, and therefore they can be accessed in a single cc. The working principle of AUs are exemplified in Figure 4.

V. EVALUATION

In this section, a comprehensive evaluation of our proposed solution is presented.

A. Implementation

The presented seven-step architecture is implemented using Verilog HDL. For configurability, Python scripts that automates generation of RTL designs are created with respect to given $n, n_{11}, n_{12}, n_{21} q$ and the desired throughput Tp. Implemented ring sizes vary from $n = 2^{10}$ to $n = 2^{16}$ for coefficient moduli $\log q = 32$ to $\log q = 64$. As previously mentioned, the target FPGA is AMD-Xilinx Alveo U280¹ (XCU280) and Vivado

¹https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

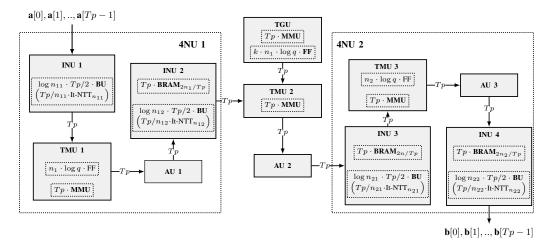


Fig. 3: Pipelined Seven-Step NTT architecture. At each cc, Tp coefficients (each of them are $\log q$ -bit) are passed to the next stage in the pipeline, represented by the solid array.

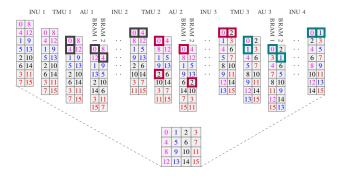


Fig. 4: A sample execution for the pipelined Seven-Step NTT architecture for n = 16, $n_{11} = n_{12} = n_{21} = n_{22} = 2$. Rows denote data in the pipeline at different points in time and between the different modules. Thick boxes highlight samples for the rotations performed by AUs, ensuring that the input of following INUs are stored in independent BRAMs.

 2023.2^2 is used for synthesis and implementation. XCU280 contains 1303680 Look-up Table (LUT)s, 2607360 FFs, 9024 DSPs³, and 2016 BRAM36E1s.

For the decomposition of n, we use a symmetric partition where $n_{11} = n_{12} = n_{21} = n_{22}$ if such a configuration is feasible. Otherwise, we prioritize assigning a larger size towards the first dimension. For example, in the case of $n = 2^{13}$, we set $n_{11} = 2^4$ and $n_{12} = n_{21} = n_{22} = 2^3$. Similarly, for $n = 2^{15}$, we assign $n_{11} = n_{12} = n_{21} = 2^4$, while $n_{22} = 2^3$. This approach is motivated by the observation that the initial blocks carry a smaller computational load compared to subsequent ones.

B. Results and Comparison

Table I summarizes comparison results with our implementations and state-of-art designs. Average latency is computed among 100 subsequent NTT operations. The Area-TimeProduct (ATP) is a metric used in literature [35], computed as Latency (μs) × (LUT + FF/2 + 100 × DSP + 300 × BRAM). Average latency is used in ATP computation since the main consideration of this study is to maximize throughput. Most of the compared implementations are based on iterative NTT architectures [13], [28], [33], [34] while some recent works are based on hierarchical approaches [11], [15]–[17], [32] as ours. In all cases, our proposed design provides better timing and ATP results. For each class, we report the design for the maximum value of Tp that is implementable in the target FPGA.

In a common setting for FHE where $n = 2^{12}$ and $\log q = 32$, our design provides $1.76 \times$ and $2.29 \times$ lower ATP compared to the existing solutions [28] and [17], respectively. The reason behind our superiority is that our solution is based on the seven-step NTT architecture which allows better throughput. Accordingly, proposed design completes an NTT in $0.52 \ \mu s$ on average, providing $4.42 \times$ speed-up. The advantage of our design is maintained in the 64-bit scenario, with an average latency improvement of $2.23 \times$ over [15] which is the solution with minimum average latency among the existing work. It is important to note that, aside from [15], none of the other existing implementations offers a fully pipelined architecture. In comparison with the iterative NTT architectures in this category, our design demonstrates $7.26 \times$ and $26.33 \times$ lower average latency compared to [13] and [33], respectively. Additionally, while accelerating NTT operation this scale, our design achieves over $1.22 \times$ better ATP than the state-of-art implementations in this category which shows that our speedoptimized design is resource efficient.

Meeting with our design goal, the effectiveness of our design scales to ring dimension from $n = 2^{10}$ to $n = 2^{16}$. Compared to the leading existing solutions for $\log q = 32$, specifically [11] for $n = 2^{13}$ and [29] for $n = 2^{14}$, our design exhibits $4.47 \times$ and $4.79 \times$ better average latency, respectively. Similarly for $\log q = 64$, our solution offers more than $4.18 \times$ speed-up for these ring sizes. Similarly, for $n = 2^{15}$ and 64-bit,

²https://www.xilinx.com/products/design-tools/vivado.html

³https://docs.amd.com/r/en-US/ug958-vivado-sysgen-ref/DSP48E2

Work	Arch.	Platform	log	q n	Tp	LUT / FF / DSP / BRAM	F	Lat. / Avg.	Avg.	ATP
							MHz	cc	μs	$.10^{-3}$
[33]	Iter.	Virtex-7	28	2^{10}	-	6.4 / 3.7 / 18 / 2	150	2113 / 1035	6.9 (26.1 x)	0.073 (1.28 x)
[26]	Iter.	XCU200	28	2^{10}	32	95 / 104 / 640 / 80	210	236	1.12 (4.25 x)	0.264 (4.60 x)
Ours	7-Step	XCU280	32	2^{10}	16	76.1 / 94.6 / 864 / 24	250	260 / 66	0.26 (1.00x)	0.057 (1.00x)
[28]	Iter.	Virtex-7	32	2^{12}	-	24.6 / 23.9 / 352 / 80	207	777	3.75 (7.22 x)	0.359 (1.76x)
[17]	4-Step	Virtex-7	32	2^{12}	-	70 / 70 / 599 / 129	200	460	2.3 (4.42 x)	0.468 (2.29x)
Ours	7-Step	XCU280	32	2^{12}	32	137 / 160 / 1632 / 40	250	404/130	0.52 (1.00x)	0.204 (1.00x)
[13] ‡	Iter.	V.Ultrascale+	60	2^{12}	-	74.5 / 61.4 / 288 / 155	250	951	3.804 (7.26x)	0.687 (1.22x)
[33]	Iter.	Virtex-7	60	2^{12}	-	21.8 / 19 / 220 / 16	150	4251 / 2070	13.8 (26.33x)	0.802 (1.43x)
[34]	Iter.	Virtex-7	60	2^{12}	-	19.1 / 17.86 / 216 / 88	270	6144	22.75 (43.43x)	1.730 (3.09x)
[35]	Iter.	Virtex-7	60	2^{12}	-	17 / 11 / 286 / 24.5	150	8937 / 4125	27.5 (52.48x)	1.607 (2.87x)
[16]	4-Step	Virtex-7	64	2^{12}	-	18.9 / 26.7 / 266 / 24	211	2490	11.80 (22.52x)	0.779 (1.39x)
[16]	4-Step	Virtex-7	64	2^{12}	-	9.2 / 12.6 / 133 / 24	241	4596	19.07 (36.39x)	0.687 (1.22x)
[15]	4-Step	XCU280	64	2^{12}	-	523 / 1478 / 6518 / 34.5	300	351	1.17 (2.23x)	2.251 (4.01x)
Ours	7-Step	XCU280	64	2^{12}	32	356.2 / 375.5 / 5040 / 72	250	464 / 131	0.52 (1.00x)	0.560 (1.00x)
[29]	Iter.	XCU280	32	2^{13}	-	29.1 / 21.5 / 224 / 64	181.8	1690	9.29 (4.47 x)	0.84 (1.60x)
Ours	7-Step	XCU280	32	2^{13}	16	89.8 / 107.9 / 1008 / 32	250	1188 / 518	2.07 (1.00x)	0.53 (1.00x)
[29]	Iter.	XCU280	32	2^{14}	-	29.1 / 21.5 / 224 / 96	181.8	3612	19.87 (4.79 x)	1.81 (1.61x)
[11] ‡	4-Step	Virtex-7	32	2^{14}	-	26.9 / 26.9 / 144 / 32.5	200	4320	21.6 (5.21x)	1.39 (1.24x)
Ours	7-Step	XCU280	32	2^{14}	16	94.2 / 112.2 / 1056 / 48	250	2224 / 1036	4.14 (1.00x)	1.12 (1.00x)
[13] ‡	Iter.	V.Ultrascale+	60	2^{14}	-	74.5 / 61.4 / 288 / 155	250	4340	17.36 (4.18x)	3.13 (1.05x)
Ours	7-Step	XCU280	64	2^{14}	16	234.1 / 255.7 / 3280 / 96	250	2292 / 1036	4.14 (1.00x)	2.97 (1.00x)
[13] ‡	Iter.	V.Ultrascale+	60	2^{15}	-	74.5 / 61.4 / 288 / 155	250	8435	33.74 (8.14x)	6.09 (1.09x)
Ours	7-Step	XCU280	64	2^{15}	32	444.6 / 459.6 / 6160 / 176	250	2300 / 1036	4.14 (1.00x)	5.56 (1.00x)
[36]	Iter.	Virtex-6	30	2^{16}	-	72.6 / 63.1 / 250 / 84	100	47795	477.95 (57.7 x)	73.77 (17.38x)
[11] ‡	4-Step	Virtex-7	30	2^{16}	-	30.8 / 36.2 / 160 / 128	196	16758	85.5 (10.32x)	8.83 (2.08x)
Ours	7-Step	XCU280	32	2^{16}	32	169.5 / 191.5 / 2016 / 152	250	4288 / 2070	8.28 (1.00x)	4.24 (1.00x)
[13] ‡	Iter.	V.Ultrascale+	60	2^{16}	-	74.5 / 61.4 / 288 / 155	250	16627	66.5 (7.96x)	12.00 (1.01x)
[33]	Iter.	XCVX485T	64	2^{16}	-	31.3 / 30 / 300 / 255	135	59400	440 (52.7 x)	67.23 (5.7 x)
[32]	3-D [†]	XCU250	64	2^{16}	-	267.1 / 328.4 / 2736 / 2126	165	62700	380 (45.5 x)	510.2 (43.3 x)
Ours	7-Step	XCU280	64	2^{16}	32	460 / 470 / 6320 / 280	248	4364 / 2070	8.34 (1.00x)	11.78 (1.00x)

TABLE I: Comparison with Literature

[†]: employs 3-D decomposition of n as $2^6 \cdot 2^6 \cdot 2^4$. [‡]: restricted to special primes.

8.15× speed-up is achieved. For the relatively smaller ring size, $n = 2^{10}$ and 32-bit, our design outperforms the literature by $4.25 \times$ in average latency. The ATP of the proposed design is slightly advantageous over the mentioned implementations, further demonstrating the resource-efficiency of our solution.

For $n = 2^{16}$, [13] is the closest competitor to our solution. However, our design achieves $7.96 \times$ improvement in the average latency while maintaining a comparable ATP. It is important to note that [13] supports only special primes, which offer advantages in resource utilization over general NTT primes. Additionally, the prime modulus in that design is fixed, preventing operation with multiple prime moduli, a key requirement in FHE applications. In contrast, our proposed design allows the prime modulus and twiddle factors to be set at run-time, providing a more practical solution for FHE scenarios. Considering more generic designs, our solution presents $10.32 \times$ and $45.5 \times$ reduction in average latency for $\log_2 q = 32$ and $\log_2 q = 64$, when compared to [11] and [32], respectively. Note that our design also outperforms these designs in terms of ATP, in accordance with the rest of the discussion.

VI. CONCLUSION

In this paper, we presented a modified four-step NTT algorithm that directly operates in the negacyclic ring $\mathcal{R}_{q,n}$. Then,

we proposed an FPGA-based hardware accelerator that implements the seven-step NTT algorithm. Our solution supports a wide-range of q and n employed by FHE. Our implementation prioritizes high throughput, low BRAM utilization, and scalability for diverse FHE applications. This architecture has been implemented on the Alveo U280 FPGA, achieving up to two orders of magnitude speed-up compared to the existing literature. Future research will focus on optimizing the resource efficiency by applying different dimensional decompositions of the ring dimension n and integrating modular reduction for special primes.

REFERENCES

- [1] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," pp. 868–886, 2012.
- [2] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." Cryptology ePrint Archive, Report 2012/144, 2012. https: //eprint.iacr.org/2012/144.
- [3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," pp. 309–325, 2012.
- [4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers." Cryptology ePrint Archive, Report 2016/421, 2016. https://eprint.iacr.org/2016/421.
- [5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," vol. 33, pp. 34–91, Jan. 2020.
- [6] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology–EUROCRYPT 2010:*, pp. 1–23, Springer, 2010.

- [7] H. L. Garner, "The residue number system," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 2, pp. 140–147, 1959.
- [8] T. Ye, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Fpga acceleration of number theoretic transform," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*, (Berlin, Heidelberg), p. 98–117, Springer-Verlag, 2021.
- [9] T.-H. Nguyen, B. Kieu-Do-Nguyen, C.-K. Pham, and T.-T. Hoang, "High-speed ntt accelerator for crystal-kyber and crystal-dilithium," *IEEE Access*, 2024.
- [10] W. Guo and S. Li, "Split-radix based compact hardware architecture for crystals-kyber," *IEEE Transactions on Computers*, 2023.
- [11] X. Chen, W. Lu, T. Su, and D. Chen, "Shp-fsntt: A scalable and highperformance ntt accelerator based on the four-step algorithm," in 2024 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5, 2024.
- [12] Z. Li, J. Ren, G. Du, Z. Tu, X. Wang, Y. Yin, and Y. Ouyang, "An area-efficient large integer ntt-multiplier using discrete twiddle factor approach," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 2, pp. 751–755, 2022.
- [13] S. Kurniawan, P. Duong-Ngoc, and H. Lee, "Configurable memory-based ntt architecture for homomorphic encryption," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 10, pp. 3942–3946, 2023.
- [14] D. H. Bailey, "Ffts in external of hierarchical memory," in *Proceedings* of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing '89, p. 234–242, Association for Computing Machinery, 1989.
- [15] Z. Guan, Y. Zhu, Y. Huang, L. Lei, X. Wang, H. Jia, Y. Chen, B. Zhang, J. Dong, and S. Bian, "Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration," in 2024 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1–6, 2024.
- [16] C. Liu, D. Tang, J. Song, H. Zhou, S. Yan, and F. Yang, "Hmntt: A highly efficient mdc-ntt architecture for privacy-preserving applications," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, GLSVLSI '24, (New York, NY, USA), p. 7–12, Association for Computing Machinery, 2024.
- [17] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2020.
- [18] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology–LATINCRYPT 2012:*, pp. 139–158, Springer, 2012.
- [19] H. Nussbaumer, "Fast polynomial transform algorithms for digital convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 2, pp. 205–215, 1980.
- [20] W. Tan, S.-W. Chiu, A. Wang, Y. Lao, and K. K. Parhi, "Parentt: Lowlatency parallel residue number system and ntt-based long polynomial modular multiplication for homomorphic encryption," *IEEE Transactions* on Information Forensics and Security, 2023.
- [21] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in 2019 IEEE International symposium on high performance computer architecture (HPCA), pp. 387–398, IEEE, 2019.
- [22] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
- [23] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, fall joint computer conference*, pp. 563–578, 1966.
- [24] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in 2019 22nd Euromicro Conference on Digital System Design (DSD), pp. 253–260, 2019.
- [25] C. Wang and M. Gao, "Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 1–9, IEEE, 2023.
- [26] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: a framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, p. 30–39, Association for Computing Machinery, 2022.
- [27] Y. Su, B.-L. Yang, C. Yang, Z.-P. Yang, and Y.-W. Liu, "A highly unified reconfigurable multicore architecture to speed up ntt/intt for

homomorphic polynomial multiplication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 8, pp. 993–1006, 2022.

- [28] S.-H. Liu, C.-Y. Kuo, Y.-N. Mo, and T. Su, "An area-efficient, conflictfree, and configurable architecture for accelerating ntt/intt," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.
- [29] C. Ayduman, E. Koçer, S. Kırbıyık, A. Can Mert, and E. Savaş, "Efficient design-time flexible hardware architecture for accelerating homomorphic encryption," in 2023 *IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–7, 2023.
- [30] D. B. Roy, D. Mukhopadhyay, M. Izumi, and J. Takahashi, "Tile before multiplication: An efficient strategy to optimize dsp multiplier for accelerating prime field ecc for nist curves," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, (New York, NY, USA), p. 1–6, Association for Computing Machinery, 2014.
- [31] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.
- [32] C. Wang and M. Gao, "Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 1–9, 2023.
- [33] F. Hirner, A. C. Mert, and S. S. Roy, "Proteus: A pipelined NTT architecture generator," 2023.
- [34] Y. Su, B. Yang, J. Wang, F. Zhang, and C. Yang, "Reconfigurable multicore array architecture and mapping method for rns-based homomophic encryption," *AEU-International Journal of Electronics and Communications*, vol. 161, p. 154562, 2023.
- [35] Z. Ye, R. C. C. Cheung, and K. Huang, "Pipentt: A pipelined number theoretic transform architecture," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, pp. 4068–4072, 2022.
- [36] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, 2018.