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

Emre Koçer
kocer@sabanciuniv.edu
Sabancı University
Istanbul, Turkey

Selim Kırbıyık
selim.kirbiyik@sabanciuniv.edu
Sabancı University
Istanbul, Turkey

Tolun Tosun
toluntosun@sabanciuniv.edu
Sabancı University
Istanbul, Turkey

Ersin Alaybeyoğlu
ersin.alaybeyoglu@sabanciuniv.edu
Sabancı University
Istanbul, Turkey

Erkay Savaş
erkays@sabanciuniv.edu
Sabancı University
Istanbul, Turkey

Abstract

Fully Homomorphic Encryption (FHE) enables computations on encrypted data, proving itself to be an essential building block for privacy-preserving applications. However, it involves computationally demanding operations such as polynomial multiplication, with the Number Theoretic Transform (NTT) being the state-of-the-art solution to perform it. Considering that 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) devices offer flexible and powerful computing platforms. We propose an FPGA-based, high-speed, parametric and fully pipelined architecture that implements the improved *Seven-Step* NTT algorithm, which builds upon the four-step algorithm. 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), which is a additional in-package memory common in high-end FGPA devices such as Alveo U280. We aim to maximize throughput through an IO parametric design on the Alveo U280 FPGA. The implementation results demonstrate that the average latency of our design for batch NTT operation is $\mathbf{8.32}\mu s$ for the ring size 2^{16} and 64-bit width; a speed-up of $\mathbf{7.96}\times$ compared to the current state-of-the-art designs.

CCS Concepts

• **Hardware** → **Hardware accelerators**; • **Security and privacy** → **Security in hardware**; • **Computer systems organization** → **Parallel architectures**.

XXXX'25, XXX, XX, XXX

ACM ISBN XXX-X-XXXX-XXXX-X/XXXX/XX
https://doi.org/XXXXXXX.XXXXXXX

Keywords

FHE, FPGA, Hardware Acceleration, Four-Step, Seven-Step, fully-pipelined, NTT, negacyclic

ACM Reference Format:
Emre Koçer, Selim Kırbıyık, Tolun Tosun, Ersin Alaybeyoğlu, and Erkay Savaş. 2025. IO-Optimized Design-Time Configurable Negacyclic Seven-Step NTT Architecture for FHE Applications. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (XXXX'25).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

FHE is an advanced cryptographic solution that allows to perform arithmetic operations, including both addition and multiplication, directly on ciphertexts without requiring decryption. This capability is particularly critical for applications such as privacy-preserving machine learning. The most practical of the existing FHE schemes found in the literature are lattice-based [3, 5, 6, 8].

which are computationally involved, rendering their deployment in real-world applications a challenging task.

The core operation in lattice-based cryptographic schemes is the multiplication of polynomials of very high degrees. Efficient implementations employ the Number Theoretic Transform (NTT) algorithm to perform this operation, with a time complexity of $O(n\log(n))$, where n refers to the degree of the polynomial modulus, which is typically between 2^{10} and 2^{16} for FHE applications. The polynomial modulus is generally the cyclotomic polynomial of the form $x^n + 1$ for FHE applications, leading to the *negacyclic* ring of $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. The coefficient modulus, a prime number used to perform modular arithmetic on the polynomial coefficients, is generally in the range of $[2^{32}, 2^{64}]$ for implementations that employ the residue number system (RNS). It is highly challenging to develop a generic approach that enables efficient hardware designs across a broad range of parameter spaces for ring dimensions as well as coefficient bit sizes. Many solutions in the literature operate only with primes of special forms to tackle some of the design challenges [4, 13, 19].

Recently, there has been a surge of interest among researchers in hierarchical NTT algorithms. These methods are based on the *four-step* approach [2], 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 approach, the *seven-step* 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 [4, 11, 14, 17]. An algorithmic drawback of the 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 pre- and post-processing steps [4, 17, 20, 21].

Except for [14], existing solutions do not offer high throughput, as they do not employ a fully pipelined architecture and concentrate only in the latency of the computation. Given that the NTT is a highly data-intensive operation for high-degree polynomials, the primary bottleneck in its computation lies in the communication between memory (or host device) and FPGA. Accordingly, throughput-oriented approaches optimized for IO bandwidth generally prove to be better than those that focus solely on minimizing latency [25, 29]. Moreover, they enable high-speed designs for multiple NTT computations, particularly in the context of FHE applications.

Our contributions to the design of hardware architectures for the fast and efficient computation of NTT can be summarized 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 also demonstrate that our novel solution directly applies to all hierarchical approaches that decompose the input polynomial into any dimension. The 4-D decomposition of the NTT input results in an improved seven-step algorithm proposed in this paper.
- Based on the improved seven-step algorithm, we present a novel, high-speed FPGA-based NTT architecture that targets FHE applications. Our solution is unique in the literature as it is fully pipelined and designed for high throughput. Our proposed architecture offers a scalable and high-performance accelerator that accommodates a wide range of parameter configurations. It achieves IO efficiency by maximizing the utilization of FPGA-host bandwidth . Our approach is adaptable to constraints arising from memory bandwidth limitations and resource utilization. Additionally, the design is configurable at design time, supporting a wide range of parameters, offering flexibility to meet different throughput and FHE parameter set requirements.
- We provide implementation results of our design on the Alveo U280 FPGA used as data center accelerator as well as on the older generation of Virtex-7 FPGA for a fair comparison with the literature. We also suggest a performance metric, *average latency*, which is the average execution time of a single NTT when many of them are executed on the pipelined architecture. This metric proves to be useful to evaluate NTT architectures within the FHE context. We demonstrate that our design scales well for high-speed computation of multiple NTT operations compared to latency-oriented designs found in the literature. The implementation results indicate that the proposed solution significantly outperforms existing work in the literature, more than an order of magnitude in terms of average latency for certain sets of parameters.

2 Background

2.1 Notation

- Lowercase italic letters denote integers, such as a. Bold uppercase letters denote matrices, such as $\mathbf{A}$. Similarly, bold lowercase letters are used to denote vectors, such as $\mathbf{a}$. Elements of matrices (vectors) are accessed using the square brackets, such as $\mathbf{A}[i][j]$ ($\mathbf{a}[i]$). $\odot$ is used to represent element-wise multiplication of vectors or matrices, such as $\mathbf{a} \odot \mathbf{b}$.
- $\mathcal{R}_{q,n}$ denotes the cyclotomic ring of polynomials $\mathbb{Z}_q[x]/(x^n+1)$. Polynomials are represented by lowercase italic letters, such as $\boldsymbol{a}(x)$. Coefficients of polynomials are represented by the sub-index, such as $\boldsymbol{a}_i$.

2.2 Number Theoretic Transform (NTT)

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

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

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 exists in $\mathbb{Z}_q$, which is denoted by ψ where $\psi^n = -1 \mod q$. For $\hat{\mathbf{a}} = \mathrm{NTT}_n(\boldsymbol{a}(x))$, the transformation is equivalent to the evaluation $\hat{\mathbf{a}}[i] = \boldsymbol{a}(\psi^{2i+1})$. Forward and backward NTT can be efficiently implemented using so-called butterfly circuits. Most applications use *Cooley-Tukey* (CT) [7] butterflies for NTT_n and *Gentleman-Sande* (GS) [10] butterflies for INTT_n. NTT algorithm utilizing CT or GS butterflies, commonly referred to as the *iterative* NTT [16, 26]. 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 $O(n^2)$.

2.2.1 Four-Step NTT. Bailey's four-step NTT, widely adopted in various studies [4, 11, 14, 17, 28], is a hierarchical approach that transforms the larger NTT into smaller and independent NTTs. Particularly, the input polynomial with n coefficients is decomposed into a matrix of size $n_1 \times n_2$. 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 1, 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 $\boldsymbol{a}(x) \in \mathbb{Z}_q[x]/(x^n - 1)$ satisfies $\hat{\mathbf{a}}[i] = \boldsymbol{a}(\omega^i)$, 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 [21]. To compute $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$ for $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathcal{R}_{q,n}$, the following pre-processing steps must be carried out:

$$\bar{a}_i = a_i \cdot \psi^i, \quad \bar{b}_i = b_i \cdot \psi^i \quad \text{for } 0 \le i < n \tag{2}$$

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:

$$c_i = \bar{c}_i \cdot \psi^{-i}, \quad \text{for } 0 \le i < n \tag{3}$$

For the iterative NTT, the pre- and post-processing steps are merged into the existing butterfly circuits by appropriately selecting the twiddle factors [22, 26]. However, for the four-step NTT, these steps are explicitly performed [9].

2.2.2 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.

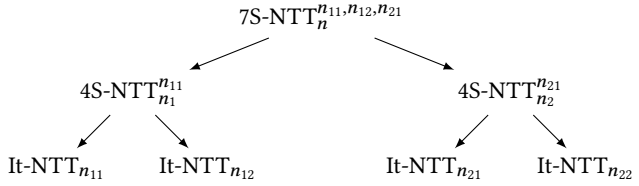

Figure 1: Illustration of seven-step NTT with recursive four-step NTTs. 7S-NTT$_n^{n_{11},n_{12},n_{21}}$ denotes seven-step NTT with 4-D decomposition $n = n_{11} \times n_{12} \times n_{21} \times n_{22}$. 4S-NTT$_n^{n_1}$ denotes four-step NTT where $n = n_1 \times n_2$ while It-NTT denotes the iterative NTT. Twiddle multiplications are not visualized.

2.3 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.

3 Negacyclic Four-Step NTT

To address the pre- and post-processing overhead associated with the four-step algorithm described in Section 2.2.1, we present a 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.

Proof: By the definition of negacyclic NTT as explained in Section 2.2, the n_1-point column NTTs block in lines 2-4 of Algorithm 1 performs the following transformation.

$$\mathbf{A}'[i][j] = \sum_{k=0}^{n_1-1} \psi^{n_2(2i+1)k} \cdot \mathbf{A}[k][j] \tag{4}$$

which is equivalent to evaluating the polynomial in the column $\mathbf{A}^T[i]$ at $\psi^{n_2 \cdot (2i+1)}$. The output of the twiddle multiplication in lines 5-9 is as follows:

$$\begin{aligned} \mathbf{A}''[i][j] &= \mathbf{A}'[i][j] \cdot \psi^{(2i-n_1+1)j} \\ \mathbf{A}''[i][j] &= \sum_{k=0}^{n_1-1} \psi^{n_2 \cdot (2i+1)k + (2i-n_1+1)j} \cdot \mathbf{A}[k][j] \end{aligned} \tag{5}$$

We plug Equation (5) into the row NTTs computed in lines 10-12:

$$\begin{aligned} \mathbf{A}'''[i][j] &= \sum_{t=0}^{n_2-1} \psi^{n_1(2j+1)t} \cdot \mathbf{A}''[i][t] \\ &= \sum_{t=0}^{n_2-1} \psi^{n_1(2j+1)t} \\ &\quad \cdot \Big(\sum_{k=0}^{n_1-1} \psi^{n_2(2i+1)k + (2i-n_1+1)t} \cdot \mathbf{A}[k][t] \Big) \\ &= \sum_{t=0}^{n_2-1} \sum_{k=0}^{n_1-1} \psi^{n_1 2jt + (2i+1)(t+n_2k)} \mathbf{A}[k][t] \end{aligned} \tag{6}$$

The output is flattened with transpose operation:

$$\hat{\mathbf{a}}[i + n_1 j] = \sum_{t=0}^{n_2-1} \sum_{k=0}^{n_1-1} \psi^{2n_1 jt + (2i+1)(t+n_2k)} \mathbf{A}[k][t] \tag{7}$$

On the other hand, the n-point NTT computes the following by definition:

$$\begin{aligned} \hat{\mathbf{a}}[z] &= \sum_{l=0}^{n-1} a[l] \psi^{(2z+1)l} \\ \hat{\mathbf{a}}[j + n_2 i] &= \sum_{t=0}^{n_2-1} \sum_{k=0}^{n_1-1} \psi^{(2(j+n_2 i)+1)(t+n_2k)} \mathbf{A}[k][t] \\ \hat{\mathbf{a}}[i + n_1 j] &= \sum_{t=0}^{n_2-1} \sum_{k=0}^{n_1-1} \psi^{(2(i+n_1 j)+1)(t+n_2k)} \mathbf{A}[k][t] \\ &= \sum_{t=0}^{n_2-1} \sum_{k=0}^{n_1-1} \psi^{(2i+1)(t+n_2k) + (2n_1 jt) + (2n_1 jn_2k)} \mathbf{A}[k][t] \\ &= \sum_{t=0}^{n_2-1} \sum_{k=0}^{n_1-1} \psi^{2n_1 jt + (2i+1)(t+n_2k)} \mathbf{A}[k][t] \end{aligned} \tag{8}$$

Note that $2n_1 n_2 = 2n = 0$ in the exponent as $\psi^{2n} = 1$. Equation (7) is equal to Equation (8). ■

Algorithm 1 Negacyclic Four-Step NTT

Input: $\boldsymbol{a}(x) \in \mathcal{R}_{q,n}$
Input: a primitive $2n$-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 $\hat{\mathbf{a}} = \mathrm{NTT}_n(\boldsymbol{a}(x))$

1: $\mathbf{A} \in \mathbb{Z}_q^{n_1 \times n_2} \leftarrow \boldsymbol{a}$ ▷ represent $\boldsymbol{a}$ as a matrix s.t. $\mathbf{A}[i][j] = \boldsymbol{a}_{i \cdot n_2 + j}$
2: **for** $i = 0 \rightarrow n_2 - 1$ **do**
3: $\quad \mathbf{A}^T[i] = \mathrm{NTT}_{n_1}(\mathbf{A}^T[i])$ ▷ using ψ^{n_2}
4: **end for**
5: **for** $i = 0 \rightarrow n_1 - 1$ **do**
6: $\quad$ **for** $j = 0 \rightarrow n_2 - 1$ **do**
7: $\quad\quad \mathbf{A}[i][j] = \mathbf{A}[i][j] \cdot \psi^{(2i - n_1 + 1)j}$
8: $\quad$ **end for**
9: **end for**
10: **for** $i = 1 \rightarrow n_1 - 1$ **do**
11: $\quad \mathbf{A}[i] = \mathrm{NTT}_{n_2}(\mathbf{A}[i])$ ▷ using ψ^{n_1}
12: **end for**
13: $\hat{\mathbf{a}} \leftarrow \mathbf{A}^T$ ▷ flatten $\mathbf{A}$ s.t. $\hat{\mathbf{a}}[j \cdot n_1 + i] = \mathbf{A}[i][j]$
14: **return** $\hat{\mathbf{a}}$

	#BUs	Twiddle Factor Storage	
		without OTF	with OTF
Iterative	$O(n \log n)$	$O(n)$	-
4-Step	$O(n^{1/2} \log n^{1/2})$	-	-
7-Step	$O(n^{1/4} \log n^{1/4})$	$O(n)$	$O(n^{1/4})^*$

*: This number is $O(\sqrt{n})$ in our implementation because of throughput requirements (see Section 4.4.2).

Table 1: Comparison of the number of Butterfly Units (BUs) needed for NTT-unrolling and the number of pre-computed twiddle factors. OTF refers to on-the-fly computation of twiddle factors for intermediate multiplication (Line 7 of Algorithm 1).

4 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 2.2.2 incorporating the negacyclic four-step approach outlined in Algorithm 1.

4.1 Design Principles

4.1.1 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.

4.1.2 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 pipelined execution with one output per clock cycle with no stalls. However, this approach demands an impractically large

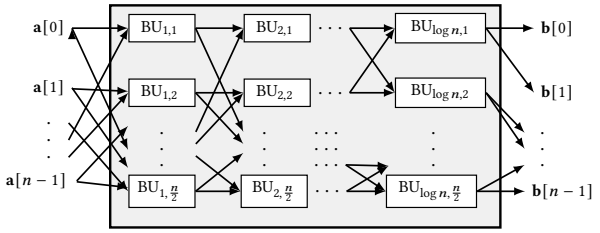

Figure 2: Loop unrolling, as illustrated for iterative NTT_n, requires $\log n \cdot (n/2)$ BUs.

number of BUs for large NTT sizes, specifically $n/2 \cdot \log n$ BUs (*e.g.*, $n = 2^{16}$ would require $2^{15} \cdot 16$ 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. In particular, the seven-step algorithm provides an improved approach for pipelined architectures for large n. Recall that there exist four independent iterative NTT operations of size $O(n^{1/4})$ in the seven-step approach, each necessitating $O(n^{1/4} \cdot \log n^{1/4})$ BUs for unrolling ($n = 2^{16}$ would require $4 \cdot 2^4/2 \cdot 4$ BUs, which is feasible with current technology). A comparison of these methods is presented in Table 1.

4.1.3 Coefficient Throughput. Our solution introduces a unique feature: a parametric hardware generator that operates based on the throughput, denoted by τ , 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. The τ parameter also defines the level of parallelism in our design. Given that $\tau \geq n_{11}, n_{12}, n_{21}, n_{22}$ and is a power-of-two, iterative NTT blocks are duplicated by τ/n_{ij} to achieve the desired throughput, as explained in Section 4.4.1.

4.1.4 Address Generation. Existing methods in the literature [1, 13, 15, 17, 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.

4.1.5 Reduced Twiddle Storage. FHE applications necessitate performing NTT with multiple primes. For instance, approximately 60 distinct primes with $\log q = 32$ and $n = 2^{16}$ 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. Classical algorithms necessitate precomputing and storing $O(n)$ twiddles. In contrast, the seven-step algorithm requires $O(n^{1/4})$ twiddle factors for the four iterative NTT executions (see Figure 1) and $O(n)$ twiddle factors for intermediate multiplications, as illustrated in Table 1. Our design employs an on-the-fly twiddle generation strategy reducing the cost of the pre-computation for the intermediate multiplications to $O(\sqrt{n})$, as detailed in Section 4.3.

4.2 Modular Multiplication

For integer multiplication, we adopt the uneven partitioning technique proposed in [23]. 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 [18] and its specialized word-level variant, the Word-Level Montgomery (WLM) [16]. 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 $q = q_H.2^{\log 2n} + 1$. 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 $2 \cdot \left\lceil \frac{64}{\log n+1} \right\rceil + 2$ clock cycles.

4.3 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 [28] to a fully pipelined design and negacyclic NTT. Let $\mathbf{p}_j[i]$ denotes $\psi^{(2\cdot i - n_1+1)\cdot j}$. Then, $\mathbf{p}_{j+1}[i] = \mathbf{p}_j[i] \cdot \mathbf{p}_1[i] = \psi^{(2\cdot i - n_1+1)\cdot (j+1)}$. 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 τ twiddle factors at each *cc* using the above explained strategy, $\mathbf{p}_1$ to $\mathbf{p}_k$ are pre-computed where $k = \lceil (\tau \cdot \mathcal{L})/n_1 \rceil$ and $\mathcal{L}$ denotes the latency of the MMU. Then, the On-the-fly Twiddle Generation Unit (TGU) outputs $\mathbf{p}_{j+k}[i']$ at each *cc*, computed based on $\mathbf{p}_j[i']$ in $\mathcal{L}$ clock cycles, for $j > k$, $\tau \cdot t \leq i' < \min(\tau \cdot (t+1), n_1)$ and some integer $t \leq \lfloor n_1/\tau \rfloor$.

4.4 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 4.3. 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 τ new coefficients as its input and produces τ coefficients as the result of the corresponding NTT operation. The throughput of all the above-mentioned units matches this value, τ . Naturally, the output stream becomes valid after a certain number of *cc*, depending on the architecture's latency.

4.4.1 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 τ and the decomposition. For instance, the INU instantiates $\tau\ /n_{11}$ iterative NTTs. For the sake of simplicity, we assume τ 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 $(n_{11}/2) \cdot log(n_{11}) \cdot \tau\ /n_{11}$ 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.

4.4.2 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 4.3). To implement the multiplication by twiddles, each TMU contains τ independently operating and pipelined MMUs to align with the τ constraint.

4.4.3 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 bit-length of the counters are distinct for each AU and depends on the parameters decomposition of n, and τ . 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

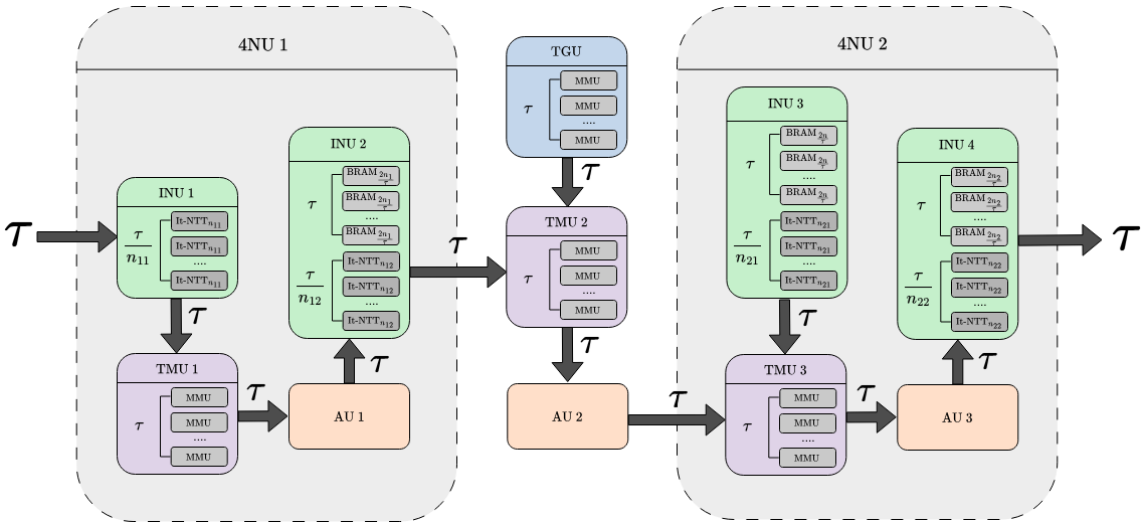

Figure 3: Pipelined Seven-Step NTT architecture. At each *cc*, τ coefficients (each of them are $\log q$-bit) are passed to the next stage in the pipeline, represented by the solid array.

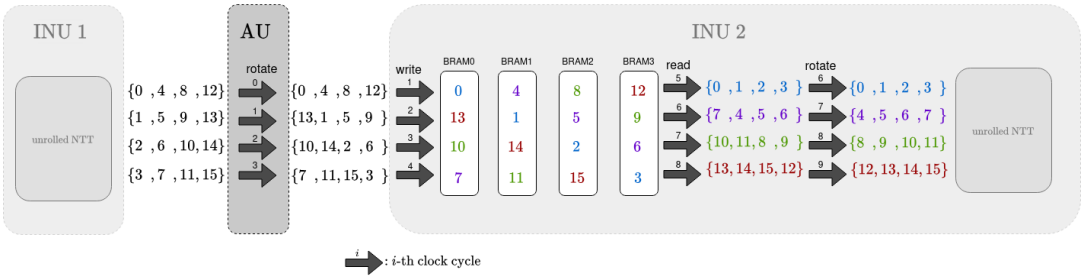

Figure 4: Operational principle of the AU is illustrated for a four-step NTT example with parameters $n = 16$, $\tau = n_1 = n_2 = 4$. The toy example contains two INUs, and single AU. TMU which is a necessary component in a four-step NTT, is omitted for simplicity. At each *cc*, the unit receives $\tau = 4$ coefficients and outputs $\tau = 4$ coefficients. Along with the BRAM read and write operations, the rotations organize the data for the subsequent INU. In this example, the preceding INU processes coefficients with a stride of 4, whereas the subsequent INU handles coefficients with a stride of 1. Note that the number of independent BRAMs is equal to τ and the BRAM status in the figure corresponds to 4^{th} *cc*.

BRAMs, and therefore they can be accessed in a single *cc*. Note that it is necessary to access the inputs of subsequent INUs within a single *cc* to maintain the pipeline. The working principle of AUs are exemplified in Figure 4.

5 Evaluation

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

5.1 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}, n_{22}, q and the desired throughput τ . Implemented ring sizes vary from $n = 2^{10}$ to $n = 2^{16}$ for coefficient moduli $\log q = 32$ to $\log q = 64$. The target FPGA devices are AMD-Xilinx Alveo U280[1] (XCU280) and VC709 FPGA, and Vivado 2023.2[2] is used for synthesis and implementation.

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 for first dimensions. 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.

Table 2: Comparison with Literature (Alveo & Ultrascale FPGAs)

Work	Arch.	Platform	$\log q$	n	τ	LUT / FF / DSP / BRAM	F (MHz)	Lat. (cc)	Avg. (μs)	ATP ($\cdot 10^{-3}$)
[29]	Iter.	XCU200	28	2^{10}	32	95 / 104 / 640 / 80	210	236	1.12 (**4.25x**)	0.264 (**4.60x**)
Ours	7-Step	XCU280	32	2^{10}	16	76.1 / 94.6 / 864 / 24	250	66	0.26 (**1.00x**)	0.057 (**1.00x**)
Ours	7-Step	XCU280	64	2^{11}	16	185 / 204 / 2640 / 40	250	130	0.52 (**1.00x**)	0.293 (**1.00x**)
Ours	7-Step	XCU280	32	2^{12}	32	137 / 160 / 1632 / 40	250	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**)
[11]	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	131	0.52 (**1.00x**)	0.560 (**1.00x**)
[1]	Iter.	XCU280	32	2^{13}	-	29.1 / 21.5 / 224 / 64	181.8	1690	9.29 (**4.47x**)	0.84 (**1.60x**)
Ours	7-Step	XCU280	32	2^{13}	16	89.8 / 107.9 / 1008 / 32	250	518	2.07 (**1.00x**)	0.53 (**1.00x**)
[1]	Iter.	XCU280	32	2^{14}	-	29.1 / 21.5 / 224 / 96	181.8	3612	19.87 (**4.79x**)	1.81 (**1.61x**)
Ours	7-Step	XCU280	32	2^{14}	16	94.2 / 112.2 / 1056 / 48	250	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	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	1036	4.14 (**1.00x**)	5.56 (**1.00x**)
Ours	7-Step	XCU280	32	2^{16}	32	169.5 / 191.5 / 2016 / 152	250	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**)
[28]	3-D†	XCU250	64	2^{16}	-	267.1 / 328.4 / 2736 / 2126	165	62700	380 (**45.5x**)	510.2 (**43.3x**)
Ours	7-Step	XCU280	64	2^{16}	32	460 / 470 / 6320 / 280	248	2070	8.34 (**1.00x**)	11.78 (**1.00x**)

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

Table 3: Comparison with Literature (Virtex-7 FPGAs)

Work	Arch.	Platform	$\log q$	n	τ	LUT / FF / DSP / BRAM	F (MHz)	Lat. (cc)	Avg. (μs)	ATP ($\cdot 10^{-3}$)
[12]	Iter.	Virtex-7	28	2^{10}	-	6.4 / 3.7 / 18 / 2	150	1035	6.9 (**18.82x**)	0.073 (**1.78x**)
Ours	7-Step	Virtex-7	32	2^{10}	16	27.5 / 41.2 / 576 / 17	180	66	0.37 (**1.00x**)	0.041 (**1.00x**)
Ours	7-Step	Virtex-7	32	2^{11}	16	30.5 / 44.4 / 624 / 17	180	130	0.722 (**1.00x**)	0.087 (**1.00x**)
[15]	Iter.	Virtex-7	32	2^{12}	-	24.6 / 23.9 / 352 / 80	207	777	3.75 (**2.60x**)	0.359 (**1.94x**)
[17]	4-Step	Virtex-7	32	2^{12}	-	70 / 70 / 599 / 129	200	460	2.30 (**1.59x**)	0.468 (**2.53x**)
Ours	7-Step	Virtex-7	32	2^{12}	16	32.4 / 47.2 / 666 / 18	180	260	1.44 (**1.00x**)	0.185 (**1.00x**)
[12]	Iter.	Virtex-7	60	2^{12}	-	21.8 / 19 / 220 / 16	150	2070	13.80 (**8.81x**)	0.802 (**1.22x**)
[27]	Iter.	Virtex-7	60	2^{12}	-	19.1 / 17.86 / 216 / 88	270	6144	22.76 (**14.52x**)	1.73 (**2.63x**)
[30]	Iter.	Virtex-7	60	2^{12}	-	17 / 11 / 286 / 24.5	150	4125	27.50 (**17.56x**)	1.61 (**2.45x**)
[14]	4-Step	Virtex-7	64	2^{12}	-	18.9 / 26.7 / 266 / 24	211	2490	11.80 (**7.53x**)	0.779 (**1.19x**)
[14]	4-Step	Virtex-7	64	2^{12}	-	9.2 / 12.6 / 133 / 24	241	4596	19.07 (**12.18x**)	0.687 (**1.05x**)
Ours	7-Step	Virtex-7	64	2^{12}	16	112 / 140 /2220 / 52	166	260	1.57 (**1.00x**)	0.657 (**1.00x**)
Ours	7-Step	Virtex-7	32	2^{13}	16	35.8 / 51.7 / 720 / 28	181	520	2.87 (**1.00x**)	0.41 (**1.00x**)
[4]	4-Step	Virtex-7	32	2^{14}	-	26.9 / 26.9 / 144 / 32.5	200	4320	21.6 (**3.74x**)	1.39 (**1.54x**)
Ours	7-Step	Virtex-7	32	2^{14}	16	39 / 55 / 768 / 44	180	1040	5.78 (**1.00x**)	0.904 (**1.00x**)
Ours	7-Step	Virtex-7	64	2^{14}	16	134 / 161 / 2560 / 104	166	1040	6.27 (**1.00x**)	3.14 (**1.00x**)
Ours	7-Step	Virtex-7	64	2^{15}	16	166 / 195 / 3120 / 170	164	2060	12.56 (**1.00x**)	7.87 (**1.00x**)
[24]	Iter.	Virtex-6	30	2^{16}	-	72.6 / 63.1 / 250 / 84	100	47795	477.95 (**20.4x**)	73.77 (**13.71x**)
[4]	4-Step	Virtex-7	30	2^{16}	-	30.8 / 36.2 / 160 / 128	196	16758	85.5 (**3.65x**)	8.83 (**1.64x**)
Ours	7-Step	Virtex-7	32	2^{16}	16	53 / 70 / 984 / 144	175	4100	23.43 (**1.00x**)	5.38 (**1.00x**)
[12]	Iter.	Virtex-7	64	2^{16}	-	31.3 / 30 / 300 / 255	135	59400	440 (**16.42x**)	67.23 (**3.58x**)
Ours	7-Step	Virtex-7	64	2^{16}	16	181 / 200 / 3264 / 310	153	4100	26.80 (**1.00x**)	18.77 (**1.00x**)

5.2 Results and Comparison

Tables 2 and 3 provide a comprehensive comparative analysis of our implementations against state-of-the-art designs. The analysis is segmented into two categories to align with the two primary classes of FPGAs: high-performance and moderate-performance devices. High-performance FPGAs, exemplified by the Alveo and Ultrascale families, typically deliver superior performance compared to their moderate-performance counterparts, such as the Virtex devices. To ensure a fair comparison, the results pertaining to moderate-performance FPGAs are evaluated against those of the Virtex-7, while the results for high-performance FPGAs are benchmarked against the Alveo U280.

Average latency is computed over the execution times of 100 subsequent NTT operations. The Area-Time-Product (ATP) is a metric used in literature [30], computed as Latency (μs) $\times$ (LUT + FF/2 + 100 $\times$ DSP + 300 $\times$ 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 [12, 13, 15, 27] while some recent works are based on hierarchical approaches [4, 11, 14, 17, 28] 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 τ that is implementable in the target FPGA.

In a standard FHE configuration with $n = 2^{12}$ and $\log q = 32$, the proposed design demonstrates **1.94**$\times$ and **2.53**$\times$ lower ATP

[1] https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
[2] https://www.xilinx.com/products/design-tools/vivado.html

compared to the existing solutions presented in [15] and [17], respectively, for Virtex-7 devices (Table 3). This performance improvement is attributed to the utilization of the NTT architecture, which facilitates higher computational throughput. As a result, the proposed architecture executes an NTT operation in an average time of 1.44 μs, achieving up to **2.60**× speed-up over the state-of-the-art solutions. The advantages of our proposed design remain evident in the 64-bit scenario with $n = 2^{12}$ on high-performance FPGAs, as shown in Table 2, achieving an average latency improvement of **2.23**× over the best design in the literature [11], which offers the minimum average latency among existing solutions. It is noteworthy that, except for [11], none of the other implementations provide a fully pipelined architecture. In comparison to iterative NTT architectures with a near 64-bit word size and $n = 2^{12}$, our design achieves **7.26**× and **8.81**× lower average latency than [13] on high-performance FPGAs and [12] on moderate-performance FPGAs, respectively. Furthermore, while accelerating the NTT operation at this scale, our design attains up to **4.01**× higher ATP compared to state-of-the-art implementations (Table 2), underscoring the resource efficiency of the proposed speed-optimized design.

Consistent with the established design objectives, the proposed solution demonstrates superior scalability across ring sizes ranging from $\boldsymbol{n = 2^{10}}$ to $\boldsymbol{n = 2^{16}}$. For $\log q = 32$, the design surpasses the state-of-the-art solutions [4] for $\boldsymbol{n = 2^{14}}$ (Table 3) and [1] for $\boldsymbol{n = 2^{13}}$ (Table 2), achieving **3.74**× and **4.47**× improvements in average latency, respectively. Furthermore, for $\log q = 64$, the solution delivers up to **4.18**× speed-up at $\boldsymbol{n = 2^{13}}$ (Table 2). At $\boldsymbol{n = 2^{15}}$ with a 64-bit word size, the proposed architecture achieves an **8.14**× speed-up on high-performance FPGAs, as shown in Table 2. For the largest ring size $\boldsymbol{n = 2^{16}}$ in FHE applications, [13] represents the closest competitor to our proposed solution for high-performance FPGA devices (see Table 2). Nevertheless, our design demonstrates a **7.96**× improvement in average latency while maintaining a comparable ATP. It should be emphasized that [13] supports only special primes, which inherently offer benefits in resource utilization compared to general NTT primes. Furthermore, the fixed prime modulus in their design restricts its applicability in operations involving multiple prime moduli, a critical aspect of FHE applications. In contrast, the proposed design introduces runtime configurability for both the prime modulus and the twiddle factors, significantly enhancing its practicality for diverse FHE scenarios. For more generic implementations, our solution achieves **3.65**× and **45.5**× reductions in average latency for $\log_2 q = 32$ and $\log_2 q = 64$, respectively, compared to [4] and [28]. Additionally, our design surpasses these alternative designs in terms of ATP, as corroborated by the broader performance analysis.

6 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, which is a direct extension of the modified four-step NTT algorithm. Our solution supports a wide-range of q and n values typically employed in practical FHE applications. Our implementation prioritizes high throughput, low BRAM utilization, and scalability for diverse FHE applications. This architecture has been implemented on the Alveo U280 FPGA and VC709 FPGA, achieving up to two orders of magnitude speed-up compared to the existing literature. Possible directions for future research would be to focus on optimizing resource efficiency by applying different dimensional decompositions of the ring dimension n and integrating modular reduction for special primes.

Acknowledgments

To Robert, for the bagels and explaining CMYK and color spaces.

References

[1] Can Ayduman, Emre Koçer, Selim Kırbıyık, Ahmet Can Mert, and Erkay Savaş. 2023. Efficient Design-Time Flexible Hardware Architecture for Accelerating Homomorphic Encryption. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*. 1–7. https://doi.org/10.1109/VLSI-SoC57769.2023.10321943

[2] D. H. Bailey. 1989. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing* (Reno, Nevada, USA) *(Supercomputing '89)*. Association for Computing Machinery, 234–242. https://doi.org/10.1145/76263.76288

[3] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. 868–886. https://doi.org/10.1007/978-3-642-32009-5_50

[4] Xiaojie Chen, Weicong Lu, Tao Su, and Dihu Chen. 2024. SHP-FsNTT: A Scalable and High-Performance NTT Accelerator Based on the Four-step Algorithm. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5.

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2016. Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Report 2016/421. https://eprint.iacr.org/2016/421.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. 33, 1 (Jan. 2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[7] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[8] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.

[9] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. 2023. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023, 4 (2023), 32–57.

[10] W Morven Gentleman and Gordon Sande. 1966. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*. 563–578.

[11] Zhenyu Guan, Yongqing Zhu, Yicheng Huang, Luchang Lei, Xueyan Wang, Hongyang Jia, Yi Chen, Bo Zhang, Jin Dong, and Song Bian. 2024. ESC-NTT: An Elastic, Seamless and Compact Architecture for Multi-Parameter NTT Acceleration. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6.

[12] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2023. Proteus: A Pipelined NTT Architecture Generator. (2023). https://eprint.iacr.org/2023/267

[13] Stefanus Kurniawan, Phap Duong-Ngoc, and Hanho Lee. 2023. Configurable memory-based NTT architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3942–3946.

[14] Changxu Liu, Danqing Tang, Jie Song, Hao Zhou, Shoumeng Yan, and Fan Yang. 2024. HMNTT: A Highly Efficient MDC-NTT Architecture for Privacy-preserving Applications. In *Proceedings of the Great Lakes Symposium on VLSI 2024*. Association for Computing Machinery, 7–12. https://doi.org/10.1145/3649476.3658734

[15] Si-Huang Liu, Chia-Yi Kuo, Yan-Nan Mo, and Tao Su. 2023. An Area-Efficient, Conflict-Free, and Configurable Architecture for Accelerating NTT/INTT. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023).

[16] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. 253–260. https://doi.org/10.1109/DSD.2019.00045

[17] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2020. Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 2 (2020), 353–362. https://doi.org/10.1109/TVLSI.2019.2943127

[18] Peter L Montgomery. 1985. Modular multiplication without trial division. *Mathematics of computation* 44, 170 (1985), 519–521.

[19] Trong-Hung Nguyen, Binh Kieu-Do-Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang. 2024. High-speed NTT Accelerator for CRYSTAL-Kyber and CRYSTAL-Dilithium. *IEEE Access* (2024).

[20] H. Nussbaumer. 1980. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28, 2 (1980), 205–215. https://doi.org/10.1109/TASSP.1980.1163372

[21] Thomas Pöppelmann and Tim Güneysu. 2012. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *Progress in Cryptology–LATINCRYPT 2012:*. Springer, 139–158.

[22] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *International conference on cryptology and information security in Latin America*. Springer, 346–365.

[23] Debapriya Basu Roy, Debdeep Mukhopadhyay, Masami Izumi, and Junko Takahashi. 2014. 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)*. Association for Computing Machinery, 1–6. https://doi.org/10.1145/2593069.2593234

[24] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. 2018. HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Trans. Comput.* 67, 11 (2018), 1637–1650.

[25] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 387–398.

[26] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact ring-LWE cryptoprocessor. In *Cryptographic Hardware and Embedded Systems–CHES 2014*. Springer, 371–391.

[27] Yang Su, Bailong Yang, Jianfei Wang, Fahong Zhang, and Chen Yang. 2023. Reconfigurable multi-core array architecture and mapping method for RNS-based homomophic encryption. *AEU-International Journal of Electronics and Communications* 161 (2023), 154562.

[28] Cheng Wang and Mingyu Gao. 2023. SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323744

[29] Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2022. NTTGen: a framework for generating low latency NTT implementations on FPGA. In *Proceedings of the 19th ACM International Conference on Computing Frontiers (CF '22)*. Association for Computing Machinery, 30–39.

[30] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. 2022. PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69 (2022), 4068–4072. https://doi.org/10.1109/TCSII.2022.3184703