ESC-NTT: An Elastic, Seamless and Compact Architecture for Multi-Parameter NTT Acceleration

Zhenyu Guan[1], Yongqing Zhu[1], Yicheng Huang[2], Luchang Lei[3], Xueyan Wang[2],
Hongyang Jia[3], Yi Chen[4], Bo Zhang[4], Jin Dong[4(✉)], Song Bian[1(✉)]
[1]School of Cyber Science and Technology, Beihang University, Beijing, China
[2]School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China
[3]Department of Electronic Engineering, Tsinghua University, Beijing, China
[4]The Beijing Academy of Blockchain and Edge Computing, Beijing, China
{guanzhenyu, zhuyongqing100, huangyicheng}@buaa.edu.cn,
llc22@mails.tsinghua.edu.cn, wangxueyan@buaa.edu.cn, hjia@tsinghua.edu.cn,
{chenyi, zhangb, dongjin}@baec.org.cn, sbian@buaa.edu.cn

***Abstract*—Fully homomorphic encryption (FHE) and post-quantum cryptography (PQC) heavily rely on number theoretic transform (NTT) to accelerate polynomial multiplication. However, most existing NTT accelerators lack flexibility when the underlying modulus and polynomial lengths change. Current designs often store twiddle factors in on-chip storage, facing a noticeable drawback when frequent parameter changes occur, leading to a potential 50% decrease in computation speed due to the input bandwidth limitations. To address this challenge, we propose ESC-NTT, a fully-pipelined and flexible architecture for handling NTTs with varying parameters. ESC-NTT, a complete custom architecture, continuously performs N-point (inverse) NTT, negacyclic NTT (NCN), and inverse NCN (INCN) without introducing bubbles during modulus and NTT length switches. Additionally, we introduce a twiddle factor generator (TFG) module to replace on-chip factor storage and save 68.7% twiddle factors' bandwidth compared to inputting every factor. In the experiment, ESC-NTT is implemented on a Xilinx Alveo U280 FPGA and synthesized in a 28 nm CMOS technology. In the case of frequent modulus switching and same on-chip storage, the calculation speed of ESC-NTT is 1.05× to 241.39× that of existing FHE accelerators when performing 4096-point NTT.**

***Index Terms*—Fully homomorphic encryption (FHE), number theoretic transform (NTT), cryptographic accelerator.**

I. Introduction

The introduction of fully homomorphic encryption (FHE) [1] in 2009 offered a secure but computationally expensive way to compute encrypted data. Over the following decade, researchers developed more efficient and practical schemes like BGV [2], B/FV [3], CKKS [4], TFHE [5], which are discussed and researched by many scholars. In FHE schemes based on ring-learning with error (RLWE) problem, high-degree polynomial multiplication over the ring is one of the most computationally intensive operations. A common optimization method is to employ number theoretic transform (NTT) butterfly circuit approach, which reduces the complexity from $O(N^2)$ to $O(N \log N)$. In addition, NTT finds widespread applications such as post-quantum cryptography (PQC) [6], and various optimizations for NTT have been proposed to accelerate multiple applications [6]–[13]. NTT is also a hotpot in security, with scholars proposing fault attacks on it [14]. However, existing FHE schemes often employ large parameters to ensure security, which makes NTT still one of the main limiting factors in calculation speed. Furthermore, NTT is frequently and extensively used in various applications of FHE. For example, each homomorphic multiplication requires six NTTs and two inverse NTTs (INTTs) in CKKS. Ren et al. [15] mentioned that packing 8192 learning with error (LWE) ciphertexts into one RLWE ciphertext requires 8191 automorphism operations, involving more than 196K 8192-point NTTs, where "point" refers to the length of a polynomial that enters the NTT circuit in a single operation. Due to the complex calculations, parallel processing and wide application of NTT in FHE, accelerators have been proposed on different hardware platforms, including FPGA, GPU, and ASIC [9], [16]–[25]. Among them, NTT implementation in FPGA or ASIC designs involves using smaller NTT butterfly circuits to process coefficients iteratively, or adopting methods like four-step NTT (2D NTT) to decompose N-point NTT circuit into N_1-point and N_2-point NTT circuits. Additionally, the FPGA accelerator design is essentially a general-purpose hardware solution and does not take advantage of the specific features of FPGA. GPU designs like TensorFHE [19] utilize a substantial amount of computational resources on GPU for acceleration, which does not apply to FPGA or ASIC designs as it can result in insufficient resources for FPGA and excessive area for ASIC, making it challenging for chip fabrication.

However, the existing accelerator's NTT circuits lack flexibility when dealing with frequently changing moduli and polynomial lengths. One of the most noticeable drawbacks is that current designs often directly store twiddle factors in on-chip storage like BRAM or URAM, such as HEAX [22], F1 [16], FAB [24]. The direct design can achieve maximum performance when processing a large amount of data with the same moduli and polynomial lengths. However, when

Song Bian, Jin Dong are the corresponding authors of this paper.

This work is partially supported by the National Key R&D Program of China (2023YFB3106200), the National Natural Science Foundation of China (62002006, 62172025, U21B2021, 61932011, 61932014, 61972018, 61972019, 62202028, U2241213). This work is also supported by Huawei Technologies Co., Ltd. and Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing.

979-8-3503-4859-0/DATE24/© 2024 EDAA

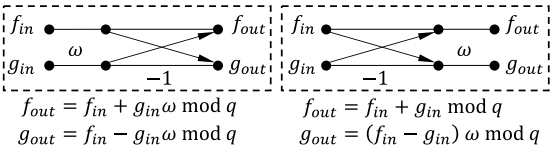

Fig. 1. (a) CT butterfly. (b) GS butterfly.

processing data with varying moduli and points continuously, it either requires additional BRAM to store different moduli and twiddle factors, or introduces extra cycles to reload BRAM, leading to additional pipeline bubbles. Moreover, scenarios involving continuous processing of data with different moduli are very common, such as performing consecutive homomorphic multiplications in NTT circuit or utilizing various moduli in residue number system (RNS) for NTT operations. For instance, when using extra cycles to reload BRAM in the case of continuously processing 1024-point NTT with different moduli in 64 cycles, each additional bubble cycle results in an approximately 1.5% decrease in efficiency. If a new modulus is required for each NTT computation, with the input bandwidth for reloading BRAM being the same as the throughput, a pipelined circuit's computation speed can decrease by 50%.

To ensure both flexibility and efficiency, we propose ESC-NTT, an Elastic, Seamless and Compact architecture for multi-parameter NTT acceleration. Inspired by four-step NTT, ESC-NTT is designed in 3D-NTT form and can be configured for different polynomial lengths. The full-pipeline design of the twiddle factor generator (TFG) module enables performing seamless (I)NTT, negacyclic NTT (NCN), and inverse NCN (INCN) when parameters switch frequently. Most existing FHE accelerators store twiddle factors on-chip, while in contrast, we design a TFG module to replace the storage and save 68.7% twiddle factors' bandwidth compared to inputting every factor. The contributions of this paper are listed as follows:

- **Elastic:** We design ESC-NTT in 3D-NTT form, where the points of all three NTT modules can be customized, and the point of the intermediate module can be switched during operation, enabling support for NTT computations with different polynomial lengths.
- **Seamless:** We observe a multiplicative or geometric progression relationship in the twiddle factors for different modules. To support fully-pipelined operations for coefficients, we also design a fully-pipelined TFG module for twiddle factors, enabling ESC-NTT to support seamless (I)NTT and (I)NCN when parameters switch frequently.
- **Compact:** 3D-NTT contributes to a smaller circuit area. We also design a TFG module to replace on-chip factor storage and save 68.7% twiddle factors' bandwidth compared to inputting every factor when $N = 1024 \sim 4096$.
- **Experiments:** ESC-NTT is implemented on a Xilinx Alveo U280 FPGA and synthesized in a 28 nm CMOS technology node. In the case of frequent modulus switching and same on-chip storage, the NTT calculation speed of ESC-NTT is 1.05× to 241.39× that of existing FHE accelerators when $N = 4096$.

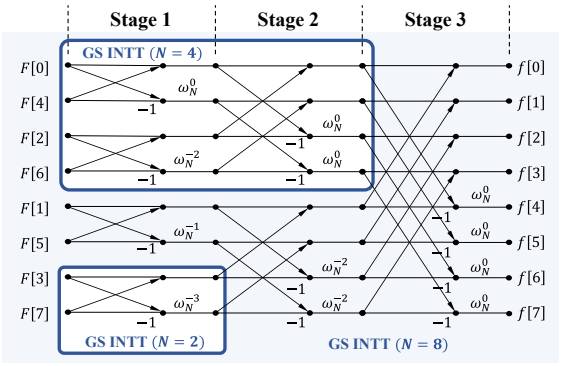

Fig. 2. GS INTT circuits ($N = 8$).

II. Preliminaries

A. RLWE-Based Homomorphic Encryption

Homomorphic encryption schemes predominantly contain two types of ciphertexts, including LWE-based and RLWE-based. In RLWE-based schemes, plaintext and ciphertext polynomials are defined over the ring $R_q = \mathbb{Z}_q[x]/(x^N+1)$, where $\mathbb{Z}_q[x]$ is a quotient ring with prime q, N is an integer which is a power of 2, and q satisfies $q \equiv 1 \bmod 2N$. To encrypt a plaintext $m(x)$ ("(x)" is omitted in some expressions), choose a random polynomial $a \leftarrow U(R_q)$, an error $e \leftarrow \Psi$ and a key $s \leftarrow \chi$, then calculate $b = -a \cdot s + \Delta \cdot m + e \bmod q$, where Δ is a larger integer less than q, and $(b, a) \in R_q^2$ is the ciphertext. Because ciphertext and plaintext are defined over the polynomial ring R_q, many operators have a significant number of (I)NTTs, as mentioned above. Therefore, accelerating (I)NTT could directly accelerate RLWE-based schemes.

B. Number Theoretic Transform and Negacyclic NTT

Number theoretic transform (NTT) can be comprehended as discrete Fourier transform (DFT) over a finite field, which is defined as Eq. (1), where $f[n]$ represents the coefficient of x^n in the polynomial $f(x)$, $F[m]$ represents x^m's in $F(x) = \mathbf{NTT}[f(x)]$, and ω_N called twiddle factor is the N^{th} primitive root of unity in $\mathbb{Z}_q$. To perform inverse NTT (INTT), the exponents of the twiddle factors need to be changed to negative values, and the result is multiplied by $N^{-1} \bmod q$.

$$F[m] = \sum_{n=0}^{N-1} f[n]\omega_N^{mn} \bmod q \tag{1}$$

NTT butterfly operations are widely adopted to accelerate polynomial multiplications over R_q, which can reduce the computational complexity from $O(N^2)$ to $O(N \log N)$. Radix-2 butterflies are divided into Cooley-Tukey (CT) and Gentleman-Sande (GS) butterflies, as shown in Fig. 1. Both of them can perform either NTT or INTT operations, and an N-point (I)NTT circuit consists of $\frac{N}{2^i}$ 2^i-point (I)NTT circuits. For example, the circuit of $N = 8$ GS INTT includes two $N = 4$ or four $N = 2$ GS INTTs, as illustrated in Fig. 2.

Negacyclic NTT (NCN) employs negative wrapped convolution (NWC) [26], an approach to avoid extensive zero-padding and simplify polynomial multiplication over the ring

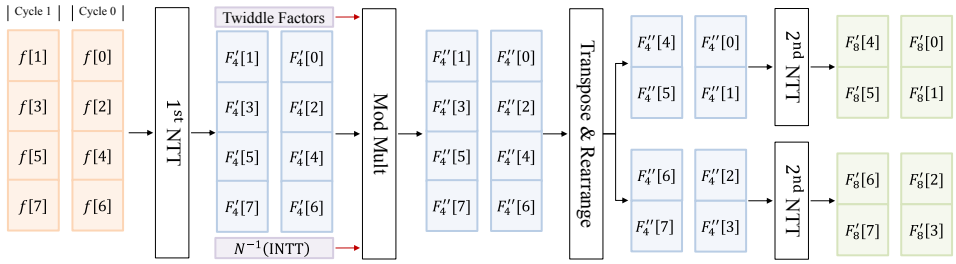

Fig. 3. Hardware-friendly four-step NTT.

Algorithm 1: NCN-Based polynomial multiplication

Input: $f(x), g(x) \in R_q, \psi_N$
Output: $h(x) = f(x) \cdot g(x), h(x) \in R_q$
1: $F(x) \leftarrow$ **NTT**$[f(x) \odot (1, \psi_N, \psi_N^2, ..., \psi_N^{N-1})]$
2: $G(x) \leftarrow$ **NTT**$[g(x) \odot (1, \psi_N, \psi_N^2, ..., \psi_N^{N-1})]$
3: $H(x) \leftarrow F(x) \odot G(x)$
4: $h(x) \leftarrow$ **INTT**$[H(x)] \odot (1, \psi_N^{-1}, \psi_N^{-2}, ..., \psi_N^{-(N-1)})]$

TABLE I
TWIDDLE FACTOR REGISTERS OF GS INCN ($N = 8$)

GS Butterfly	Stage 1	Stage 2	Stage 3
Butterfly 1	$\psi_N^{-1}\omega_N^0$	$\psi_N^{-2}\omega_N^0$	$\psi_N^{-4}\omega_N^0$
Butterfly 2	$\psi_N^{-1}\omega_N^{-2}$	$\psi_N^{-2}\omega_N^0$	$\psi_N^{-4}\omega_N^0$
Butterfly 3	$\psi_N^{-1}\omega_N^{-1}$	$\psi_N^{-2}\omega_N^{-2}$	$\psi_N^{-4}\omega_N^0$
Butterfly 4	$\psi_N^{-1}\omega_N^{-3}$	$\psi_N^{-2}\omega_N^{-2}$	$\psi_N^{-4}\omega_N^0$

R_q. NCN-based polynomial multiplication is described by Algorithm 1, where $\odot$ denotes component-wise multiplication, and ψ_N is the $2N^{th}$ primitive root of unity in $\mathbb{Z}_q$, such that $\psi_N^2 = \omega_N$. For convenience, we refer to both ψ_N and ω_N as twiddle factors. The implementation of NCN proposed by Pöppelmann et al. [27] incorporates ψ_N of NCN into a CT NTT circuit, and ψ_N^{-1} of inverse NCN (INCN) into a GS INTT circuit, which saves $N/2$ multipliers for each, respectively. The twiddle factors used in GS INCN ($N = 8$) are shown in Table I, are similarly employed in the implementation of CT NCN.

C. Four-Step NTT

Four-step NTT is a method of decomposing an N-point NTT circuit into an N_1-point and an N_2-point NTT circuit, where $N = N_1 \times N_2$. To accommodate hardware constraints and ensure consistent input-output throughput, the improved four-step NTT primarily consists of four steps: 1^{st} NTT, Modular Multiplication (Mod Mult), Transpose & Rearrange (TR), and 2^{nd} NTT. The dataflow of decomposing an 8-point NTT can be illustrated in Fig. 3. By this way we can decompose an N-point NTT circuit for any N into N_1-point, N_2-point, ..., N_i-point NTT circuits, where $N = N_1 \times N_2 \times ... \times N_i$, effectively reducing the area of an NTT circuit.

III. ESC-NTT ARCHITECTURE

A. Overview

The architecture of ESC-NTT is shown in Fig. 4. Inspired by four-step NTT, fully-pipelined ESC-NTT is designed in 3D-NTT form, composed of NTT modules, Transpose & Rearrange (TR) modules, Modular Multiplication (Mod Mult) modules and Twiddle Factor Generator (TFG) modules. When ESC-NTT is operating, coefficients and twiddle factors are fetched from Memory and loaded into the circuit for N-point NTT calculations following the dataflow depicted in Fig. 4. Memory can be either DDR or HBM based on the input bandwidth requirements, and NTT modules perform small-parameter NTT operations on coefficients. TR modules handle coefficient transposition and rearrangement, and Mod Mult modules multiply each coefficient by its twiddle factor. TFG modules supply the correct twiddle factors to the modules requiring them. In addition, INTT is achieved by adjusting twiddle factors and multiplying N^{-1} within the 2^{nd} Mod Mult module. (I)NCN is realized and will be elaborated in Sec. III-E.

B. NTT Module

ESC-NTT consists of NTT modules: one CT NTT module and two GS NTT modules. For a comprehensive representation of NTT settings, we depict the points of NTTs from input to output as a vector (N_1, N_2, N_3), where $N_1 = N_3, N = N_1N_2N_3$. For instance, a 2048-point NTT can be denoted as $(16, 8, 16)$. The first and third NTT modules are fixed-point NTTs, while the intermediate GS NTT can be configured, following the theory in Sec. II-B, as $\frac{N_2}{2^i}$ 2^i-point NTTs depending on the requirements. As the 2^{nd} GS NTT excludes (I)NCN implementations, the last stage's multiplier inputs are either multiplied by a twiddle factor ω^0 or bypassed when $N_2 \neq N_1$, rendering the multipliers negligible. Meanwhile, the 1^{st} and 3^{rd} NTTs implement (I)NCN following an approach similar to Pöppelmann's [27] as explained in Sec. II-B, which results in that the multipliers in stage 1 of the CT NTT and the last stage of the GS NTT cannot be disregarded. The (I)NCN will be elucidated in Sec. III-E.

C. Transpose & Rearrange Module

Two Transpose & Rearrange (TR) modules are incorporated into the circuit, denoted as the 1^{st} and the 2^{nd} TR module. In the 2^{nd} one, a traditional crossbar method is employed to achieve the transpose operation from $N_2 \times N_3$ to $N_3 \times N_2$ for every N_2N_3 coefficients with registers and a finite-state machine (FSM), where ESC-NTT is configured as (N_1, N_2, N_3).

Due to the necessity of performing the transpose and rearrangement from $N_2N_3 \times N_1$ to $N_1 \times N_2N_3$ in the 1^{st} TR module, implementing a crossbar with registers is impractical due to its extensive resource consumption. Therefore, we implement the 1^{st} TR module by First-In-First-Outs (FIFO) constructed with Block RAMs (BRAM). Taking inspiration from the crossbar, we constructed a $N_1 \times N_3$ FIFO array using FIFOs with a depth of $2N_1N_3$ and a width of 64 bits.

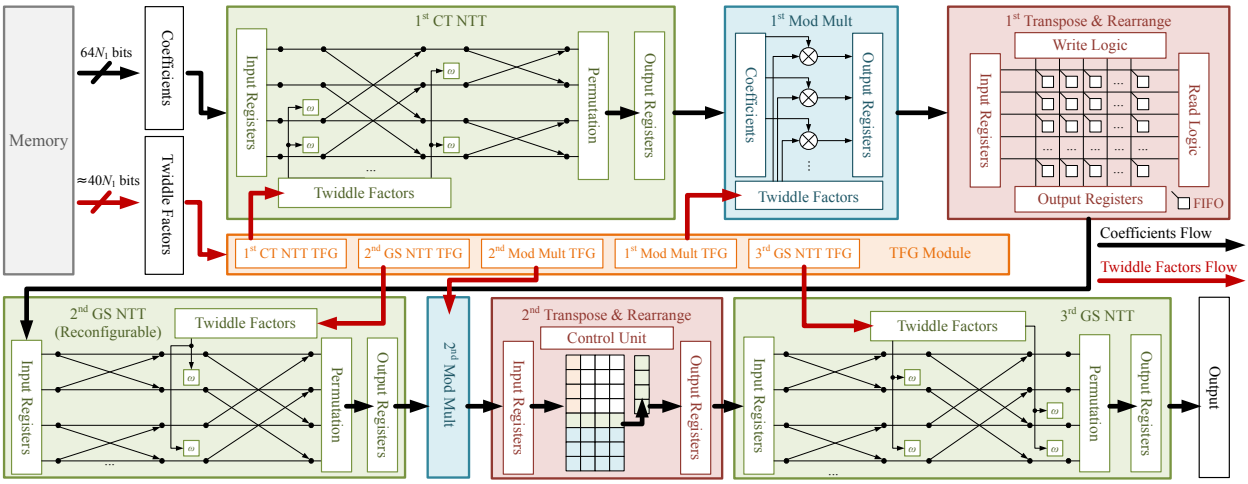

Fig. 4. Overall architecture.

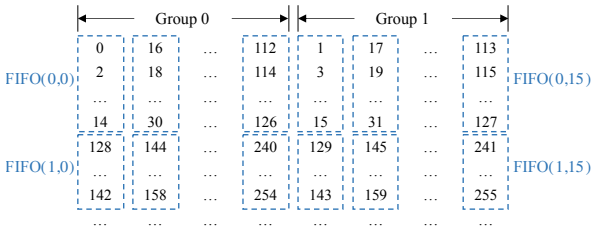

Fig. 5. Data in FIFO array when $N = 2048$.

Subsequently, we will explain how the 1^{st} TR module operates. We employ a coordinate system in the format of $(row, column)$ to locate a FIFO block. For example, the FIFO at the top left corner of the array is labeled as (0,0) in Fig. 4. Through theoretical derivation, we find that at the c_i-th cycle, the input to the 1^{st} TR module comprises N_1 numbers from the N input coefficients, indicated as Eq. (2), where element v_i represents the v_i-th coefficient of the input polynomial.

$$(c_i, c_i + N_2N_3, ..., c_i + (N_1 - 1)N_2N_3) \tag{2}$$

For the same N coefficients, the outputs in the c_o-th cycle can be expressed by Eq. (3). Except for the first element, the rest are denoted by $+i$, meaning adding i to the initial element. When $N_2 \neq N_1$, the rearrangement operation involves partitioning N_1 coefficients into different groups marked by G per cycle, enabling N_2-point NTT on each group for the 2^{nd} NTT calculation. In the example of $(16, N_2, 16)$, $G = 0, 1, 2, 3$ when $N_2 = 4$, and $G = 0, 1$ when $N_2 = 8$. The module finally outputs an N_3-length vector per cycle, which is sequentially combined by these groups. Later the 2^{nd} TR will rearrange them to the correct sequence for the 3^{rd} NTT calculation.

$$(N_2N_3\lfloor\frac{c_o}{N_2}\rfloor + \frac{N_3}{N_2}(c_o \bmod N_2) + G, +N_3, ..., +(N_2-1)N_3) \tag{3}$$

For example, considering $N = 2048$ and assuming that FIFOs are initially empty, these 2048 coefficients will be placed into the FIFO array following the pattern illustrated in Fig. 5. After approximately 256 cycles of waiting for data input, the output retrieval process commences from the first FIFO row of the array. Each cycle, the module will output N3 data from the top of this row. Once all the data related to these N coefficients have been outputted from the first row, the process switches to the next FIFO row for further output.

D. Modular Multiplication Module

Modular reduction is the most expensive and frequent essential operation in NTT. In ESC-NTT, every modular reduction is implemented with a proposed NTT-friendly Montgomery Reduction by Mert et al. [28]. The 1^{st} and the 2^{nd} Mod Mults in Fig. 4 are identical modules, each comprising N_1 modular multipliers, meaning each has N_1 twiddle factors or variations as inputs. Theoretical analysis reveals that these N_1 numbers change every cycle, which results in a substantial bandwidth overhead when on-chip storage is not used to store twiddle factors. Consequently, the TFG module is introduced to replace on-chip storage for inputting twiddle factors and their variants.

E. Twiddle Factor Generator Module

Twiddle factor generator (TFG) module is the key to realizing seamless (I)NTT or (I)NCN without bubble. Additionally, owing to TFG, the twiddle factor bandwidth has been reduced to an acceptable range. ESC-NTT has five modules using twiddle factors. Hence the TFG module comprises five units, each dedicated to one module. Before explaining TFG units in detail, the implementation of (I)NCN in ESC-NTT will be elucidated, which significantly impacts the TFG design.

NCN is realized by the 1^{st} CT NTT and the 1^{st} Mod Mult, while INCN is realized by the 2^{nd} Mod Mult and the 3^{rd} GS NTT. At the c_i-th cycle, the input to the 1^{st} CT NTT comprises N_1 numbers as a vector whose elements are consistent with the expression in terms of Eq. (2). When performing NCN, we need to perform component-wise multiplication between this vector and vector Eq. (4) associated with ψ_N. we find that the exponents of ψ_N in vector Eq. (4) form an arithmetic progression with a difference of N_1N_2, and in the c_i-th cycle, there is an offset of c_i for the ψ_N exponents of all coefficients.

$$(\psi_N^{c_i}, \psi_N^{c_i+N_2N_3}, ..., \psi_N^{c_i+(N_1-1)N_2N_3}) \tag{4}$$

TABLE II
THE INPUT TWIDDLE FACTORS AND OUTPUT STAGES OF FIVE TFG UNITS ($\psi_N^{-1} = 1$ WHEN NCN, $\psi_N = 1$ WHEN INCN, BOTH $= 1$ WHEN (I)NTT

TFG Unit	Input Factors	Output Stage
1^{st} CT NTT	$\omega_{16}, \omega_{16}^4$ or $\psi_N^{16N_2}, \psi_N^{64N_2}, \psi_N^{128N_2}$	0,1,2,3
1^{st} Mod Mult	$\omega_N^{c_i}, \psi_N^{c_i}, (\omega_N \psi_N)^{c_i}$	4
2^{nd} GS NTT	ω_{N_2}	3
2^{nd} Mod Mult	$\omega_{16N_2}, \omega_{16N_2}^{\frac{16c_i}{N_2}}, \psi_N^{-16}, N^{-1}\psi_N^{-\lfloor \frac{c_i}{N_2} \rfloor}$	6
3^{rd} GS NTT	$\omega_{16}, \psi_N^{-N_1 N_2}$	4,5,6,7

Therefore, we can make slight modifications to the twiddle factors of the 1^{st} CT NTT employing Pöppelmann's method [27] by replacing every original ψ_N^i with $\psi_N^{i \times N_2 N_3}$, and in the 1^{st} Mod Mult, multiplying the twiddle factors for the c_i-th cycle by $\psi_N^{c_i}$. As a result, except for the stage 1 butterfly in the 1^{st} CT NTT, which cannot omit the modular multipliers, no adjustments are needed for the rest of the circuit structure.

The implementation of INCN is similar to that of NCN, which involves modifying the 3^{rd} GS NTT by replacing every original ψ_N^{-i} with $\psi_N^{-i \times N_2 N_3}$ in a twiddle factor table like TABLE I. Additionally, because there is a TR module before the 3^{rd} GS NTT, the twiddle factors for the 2^{nd} Mod Mult need component-wise multiply a vector V composed of $\frac{N_3}{N_2}$ identical vectors V_1 in Eq. (5), rather than $\psi_N^{-c_i}$.

$$V_1 = (\psi_N^{-\lfloor \frac{c_i}{N_2} \rfloor}, \psi_N^{-\lfloor \frac{c_i}{N_2} \rfloor - N_3}, ..., \psi_N^{-\lfloor \frac{c_i}{N_2} \rfloor - (N_2-1)N_3}) \quad (5)$$

The concept employed by TFG is straightforward: generating ω^is from a single ω. Therefore, TFG units are cascaded with multiple sets of modular multipliers and MUXes, where each set is also referred to as a stage. The largest stage number indicates the number of multiplier sets contained within the TFG units. We observe that the twiddle factors used in NTT modules are in multiple relationships, while the twiddle factors within each cycle for the input of N_1 or N_3 numbers used in both two Mod Mult modules exhibit geometric progressions.

Due to the proximity of the 1^{st} CT NTT and the 1^{st} Mod Mult to the beginning of the circuit, rigorous design of the internal data flow in those two TFG units is required to ensure that the twiddle factors are promptly delivered to the circuit for computation, preventing an increase in circuit delay. This means that both modules require more than one twiddle factor as their TFG input bases. Furthermore, in the 1^{st} CT NTT TFG, the twiddle factors are generated in a timely manner based on the stage of the NTT, for the unit need to generate different twiddle factors for different NTT stages. The design of the 2^{nd} GS NTT TFG is relatively straightforward as it only needs to generate a sequence $\omega_{N_2}^2 \sim \omega_{N_2}^7$ from ω_{N_2}. The 2^{nd} Mod Mult TFG is more complex due to the need to consider different geometric progressions for various N_2 output lengths and to support INCN, which involves considering ψ_N^{-1} and N^{-1}. The design of the 3^{rd} GS NTT is similar to that of the 2^{nd} one, with the additional consideration of $\psi_N^{-N_1 N_2}$.

The input twiddle factors and output stages of these five TFG units in an ESC-NTT implementation, which is set in $(16, N_2, 16)$ where $N_2 = 4, 8, 16$, are illustrated in Table II, where c_i represents the data entering the unit in the c_i-th cycle. It can be observed that the inputs involving c_i change every cycle, resulting in an unavoidable bandwidth requirement. In this example, each cycle requires at least four inputs with c_i and one constant input (five in total), excluding the bandwidth of coefficients, with a bandwidth requirement of 11.2 GB/s.

IV. EXPERIMENTAL RESULTS

A. Experiment Setup

In FHE, the level of security diminishes for values of N less than 1024. Therefore, ESC-NTT implementation is set to support N-point NTT with $N = 1024, 2048, 4096$ and 64-bit q, making it versatile for various applications and resource-sufficient for FPGA implementation. Using the SEAL library, the single-core CPU computation speed is tested by an Intel(R) Xeon(R) Gold 6226R CPU @2.90GHz.

B. Implementation Results

For FPGA, ESC-NTT implemented on a Xilinx Alveo U280 FPGA is designed in Verilog and synthesized as an RTL-kernel using Vitis 2021.2, achieving $3.813\mu s$ delay and a clock frequency of 300MHz. ESC-NTT utilizes 523K LUTs, 1,478K FFs, 6518 DSPs and 1237.5KB BRAMs. The estimated bandwidth of twiddle factors is 11.2 GB/s; meanwhile, inputting every twiddle factor requires 35.8 GB/s (at least 16 64-bit integers per cycle). The total on-chip power of FPGA implementation is 82.77W. Additionally, ESC-NTT is synthesized in a 28nm CMOS technology, realizing a frequency of 500MHz, a power consumption of 47.745mW, and an ideal area of $0.082mm^2$, with an estimated actual area of $0.131mm^2$.

C. Performance

We compared the NTT accelerator's performance across different platforms, and the results are shown in Table III, where "TF" means twiddle factor, and "Converted Speed" represents the number of NTTs performed when the max.bit is 54 (This requires the circuits whose max.bit less than 54 to perform NTT twice for the same polynomial in certain parameter sets to achieve the desired security level) and some special scenarios will be mentioned below. In comparison to CPU and GPU performance under ideal conditions, ESC-NTT on FPGA demonstrates speedups ranging from 1.29× to 52.73×. For ASIC, F1 and ARK can calculate faster when $N = 4096$ because ESC-NTT's area is only $0.131mm^2$, whereas F1's NTT area is $2.27mm^2$, and ARK's is $14.3mm^2$. If ESC-NTT were scaled up to the same area and max.bit, it could achieve 2.17 times the speed of F1 and 3.41 times the speed of ARK. For fairness, we synthesized ESC-NTT in (32,32,32) to compare performance when $N = 16384$, with an area of 2.10mm². The converted speed of ESC-NTT is 0.27× and 0.42× that of F1 and ARK. However, when the max.bit is set to 64 bits, in contrast to F1 and ARK, which both require approximately 2 TB/s of input bandwidth for coefficients and twiddle factors, ESC-NTT is more practical, requiring only about 140 GB/s. Some schemes

TABLE III
PERFORMANCE COMPARISON OF NTT ($N = 4096$) ACROSS VARIOUS ACCELERATORS AND UTILIZATION OF TF STORAGE

Design	Platform	Max.bit	Freq. (MHz)	NTT/second	BRAM(KB) or Area(mm^2)	Converted Speed (Speedup)	TF Optimized	No Requirement for TF Storage
Intel Xeon 6226R	CPU	54	-	22,223	-	-	-	-
P100 [18]	GPU	32	-	27,777	-	-	-	-
TensorFHE [19]	GPU	54	-	910,134	-	-	-	-
ESC-NTT(Ours)	FPGA	64	300	1,171,875	1237.5	**947 (1.00×)**	✓	✓
F1 [16]	ASIC 14/12nm	32	1000	31,250,000	2.27	6,883,260 (2.17×)	✗	✗
ARK [17]	ASIC 7nm	64	1000	62,500,000	14.3	4,370,629 (3.41×)	✓	✗
ESC-NTT(Ours)	ASIC 28nm	64	500	1,953,125	0.131	**14,909,351 (1.00×)**	✓	✓
Roy [20]	FPGA	30	200	13,699	1746	4 (241.39×)	✗	✗
HEAX [22]	FPGA	40	300	195,313	1813	36 (26.37×)	✗	✗
Medha [23]	FPGA	54	200	130,209	3344	39 (24.32×)	✓	✗
FAB [24]	FPGA	54	300	3,125,000	6912	301 (3.14×)	✗	✗
Poseidon-NDP LBHR [25]	FPGA	40	300	1,171,875	1044	449 (2.11×)	✓	✗
Poseidon-NDP HBLR [25]	FPGA	40	300	2,343,750	1044	898 (1.05×)	✗	✗
ESC-NTT(Ours)	FPGA	64	300	1,171,875	1237.5	**947 (1.00×)**	✓	✓

optimize twiddle factor input for FPGA accelerators, while others do not. For schemes whose twiddle factor input is not optimized, assuming that reloading twiddle factors uses the same input bandwidth and BRAM reloading is required every two (I)NCN computations, resulting in a speed reduction to 2/3. Additionally, one advantage of ESC-NTT lies in on-chip storage compression. Therefore, for FPGA, we calculate the speed in the case of frequent parameter switching and then divide it by their respective BRAM sizes to obtain the converted speed. In conclusion, ESC-NTT's converted speed outperforms the mentioned works by a factor of 1.05 to 241.39.

V. CONCLUSION

We propose ESC-NTT, a customizable solution capable of (I)NTT and (I)NCN operations without introducing bubbles during continuous parameter switching. We design a TFG module to replace on-chip twiddle factor storage and save 68.7% twiddle factors' bandwidth compared to inputting every factor. Additionally, ESC-NTT is implemented on a Xilinx Alveo U280 FPGA and synthesized in a 28 nm CMOS technology. In the case of frequent modulus switching and same on-chip storage, the calculation speed of ESC-NTT is 1.05× to 241.39× that of existing FHE accelerators when performing 4096-point NTT.

REFERENCES

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.
[2] Z. Brakerski *et al.*, "(Leveled) Fully Homomorphic Encryption Without Bootstrapping," in *ITCS*, 2012.
[3] J. Fan *et al.*, *Somewhat Practical Fully Homomorphic Encryption*, IACR Cryptol. ePrint Arch., 2012.
[4] J. H. Cheon *et al.*, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *ASIACRYPT*, 2017.
[5] I. Chillotti *et al.*, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," *J. Cryptol.*, 2020.
[6] E. Karabulut *et al.*, "RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform," in *FPL*, 2020.
[7] A. C. Mert *et al.*, "An Extensive Study of Flexible Design Methods for the Number Theoretic Transform," *IEEE TC*, 2022.
[8] A. C. Mert *et al.*, "A Flexible and Scalable NTT Hardware : Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography," in *DATE*, 2020.
[9] X. Chen *et al.*, "CFNTT: Scalable Radix-2/4 NTT Multiplication Architecture with an Efficient Conflict-free Memory Mapping Scheme," *TCHES*, 2022.
[10] X. Chen *et al.*, "Efficient access scheme for multi-bank based NTT architecture through conflict graph," in *DAC*, 2022.
[11] Y. Itabashi *et al.*, "Efficient Modular Polynomial Multiplier for NTT Accelerator of Crystals-Kyber," in *DSD*, 2022.
[12] N. Zhang *et al.*, "Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT," *TCHES*, 2020.
[13] Y. Wei *et al.*, "The-v: Verifiable privacy-preserving neural network via trusted homomorphic execution," in *ICCAD*, 2023.
[14] P. Ravi *et al.*, "Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform," *TCHES*, 2023.
[15] X. Ren *et al.*, "HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database," *VLDB*, 2022.
[16] N. Samardzic *et al.*, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in *MICRO-54*, 2021.
[17] J. Kim *et al.*, "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," in *MICRO-55*, 2022.
[18] A. Al Badawi *et al.*, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *TCHES*, 2018.
[19] S. Fan *et al.*, "TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU," in *HPCA*, 2023.
[20] S. Sinha Roy *et al.*, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," in *HPCA*, 2019.
[21] F. Turan *et al.*, "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA," *IEEE TC*, 2020.
[22] M. S. Riazi *et al.*, "HEAX: An Architecture for Computing on Encrypted Data," in *ASPLOS*, 2020.
[23] A. C. Mert *et al.*, *Medha: Microcoded Hardware Accelerator for computing on Encrypted Data*, arXiv, 2022.
[24] R. Agrawal *et al.*, "FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption," in *HPCA*, 2023.
[25] Y. Yang *et al.*, "Poseidon-NDP: Practical Fully Homomorphic Encryption Accelerator Based on Near Data Processing Architecture," *TCAD*, 2023.
[26] P. Longa *et al.*, "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography," in *CANS*, 2016.
[27] T. Pöppelmann *et al.*, "High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers," in *LATINCRYPT*, 2015.
[28] A. C. Mert *et al.*, "Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture," in *DSD*, 2019.