A Run-Time Configurable NTT Architecture for Homomorphic Encryption Based on 3D Algorithm

Weicong Lu, Xiaojie Chen, Dihu Chen and Tao Su*
School of Electronics and Information Technology (School of Microelectronics)
Sun Yat-Sen University
Guangzhou, China
{luwc3, chenxj86}@mail2.sysu.edu.cn, {stscdh, sutao}@mail.sysu.edu.cn

***Abstract*—Homomorphic encryption (HE) allows computations on encrypted data without compromising data privacy, making it ideal for scenarios like privacy-preserving computing. The primary bottleneck within HE schemes is polynomial multiplication, which can be accelerated using the number theoretic transform (NTT). This paper proposes a run-time configurable (RTC) NTT/INTT accelerator supporting HE parameter sets based on the 3D NTT algorithm. A conflict-free memory access pattern is proposed to efficiently implement the 3D NTT algorithm without additional hardware units. Additionally, an on-the-fly twiddle factor generator (TFG) is proposed to optimize memory utilization for twiddle factors (TFs). The proposed design achieves significant improvements in performance and area efficiency compared to state-of-the-art FPGA implementations.**

***Index Terms*—Homomorphic encryption, run-time configurable, number theoretic transform, FPGA.**

I. INTRODUCTION

Homomorphic encryption (HE) is a cryptographic scheme that enables computations directly on ciphertexts without compromising data privacy, making it suitable for scenarios like privacy-preserving computing. Most HE schemes are based on the ring learning with errors (RLWE) problem, involving a large number of polynomial operations over the ring, with polynomial multiplication being the primary bottleneck [1]. The number theoretic transform (NTT) is considered as the current dominant approach in solving the polynomial multiplication bottleneck in HE. However, existing NTT accelerators suffer from two limitations. Firstly, for practical applications, the polynomial degree of HE typically ranges from 2^{15} to 2^{18}, and the modulus can be up to 64 bits [2]. Nevertheless, most NTT accelerators only support small parameter sets, and scaling them to HE results in performance degradation and resource utilization increasing. Secondly, the parameters in HE vary significantly across different schemes and applications, creating a demand for dynamically configurable NTT accelerators that support a wide range of parameter sets without recompilation, known as run-time configurable (RTC) NTT accelerators [3]. However, most existing works support either fixed parameter sets or demonstrate optimal performance only within relatively narrow parameter set ranges.

Traditional NTT architectures can be categorized into two types: iterative NTT architecture [4]–[8] and pipelined NTT architecture [9]–[12]. The iterative NTT architecture employs multiple parallel processing elements (PEs) and memory blocks to perform the butterfly operations for each stage, iterating these PEs to complete all stages. However, for large polynomial degrees, supporting RTC with the iterative NTT architecture introduces complex inter-stage data dependencies and increased control logic overhead, which leads to increased resource consumption and lower clock frequency. In contrast, the pipelined architecture completely unrolls the NTT, enabling RTC NTT by bypassing certain pipeline stages. Nevertheless, this approach requires sufficient resources to support the highest polynomial degree, and excessive bypassing of pipeline stages may result in significant resource underutilization. Additionally, the pipelined NTT architecture suffers from limited throughput, with most implementations can only process one or two inputs and outputs per clock cycle.

Recently, there has been a trend of utilizing multi-dimensional algorithms in NTT hardware designs. Most works adopt 2D NTT [13]–[15], and a few of them explore higher dimensions [16], [17]. Multi-dimensional NTT represents a one-dimensional polynomial coefficient vector as a multi-dimensional tensor, decomposing the large-point NTT into independent small-point NTTs along each tensor dimension. Such decomposition enables efficient reuse of a small-point NTT processing unit (PU) across different tensor dimensions. The simplification of PU greatly mitigates the complex data dependencies between NTT stages and reduces resource under-utilization in RTC. Moreover, the independence of the small-point NTTs suggests a greater potential for parallelism. These benefits render multi-dimensional NTT an optimal choice for RTC NTT accelerators supporting HE parameter sets. However, multi-dimensional NTT also introduces additional challenges. Transitioning from a one-dimensional vector to a multi-dimensional tensor complicates memory access, often requiring additional hardware units, such as transpose units to implement the algorithms without conflict. Furthermore, twiddle factor (TF) multiplications are needed when the dimension is switched, necessitating additional memory resources.

This paper focuses on 3D NTT and proposes a high-performance and area-efficient NTT/INTT accelerator that supports RTC within HE parameter sets. To overcome challenges introduced by 3D NTT, several novel optimizations are applied. The contributions are specified as follows:

(1) An NTT/INTT accelerator based on the 3D NTT algo-

rithm that supports RTC within HE parameter sets is proposed. A pipelined NTT (PNTT) module is designed as the core component of PU for small-point NTTs/INTTs.

(2) The conflict-free memory access pattern proposed in our previous work [18] is extended to efficiently implement the 3D NTT algorithm without additional hardware units. This approach reduces the extra hardware cost induced by memory access while maintaining throughput.

(3) A twiddle factor generator (TFG) is proposed to generate the TFs on-the-fly for dimension switching. By storing multiple TF seeds, this unit can generate TFs for NTT/INTT across various polynomial degrees.

II. Background

A. NTT-based Polynomial Multiplication

In this paper, the polynomial multiplication in HE is defined over the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^N + 1)$, where q is a prime modulus satisfying $q \equiv 1 (\mathrm{mod}\ 2N)$ and N is a power of two, representing the polynomial degree. The process of polynomial multiplication over R_q takes two polynomials $a(x) = \sum_{i=0}^{N-1} a_i x^i$ and $b(x) = \sum_{i=0}^{N-1} b_i x^i$ as inputs and produces an output polynomial $c(x) = a(x) \cdot b(x)$. Subsequently, the product is reduced by modulo $(x^N + 1)$ and each coefficient is further reduced by modulo q. NTT-based polynomial multiplication converts convolution into coefficient-wise multiplication, reducing the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nlog_2 N)$. Through additional pre-processing and post-processing steps, the technique negative wrapped convolution (NWC) [19] can further eliminate the reduction with modulo $(x^N + 1)$. [20] has proposed merging pre-process and post-process with Cooley-Tukey (CT) butterfly and Gentleman-Sande (GS) butterfly, respectively. In this work, the merged algorithms are adopted to further eliminate the costly pre-processing and post-processing steps.

B. 3D NTT Algorithm

The 3D NTT algorithm represents a one-dimensional polynomial coefficient vector of length N as a three-dimensional tensor A_0 of dimension $n_1 \times n_2 \times n_3$, where $N = n_1 \times n_2 \times n_3$. Specifically, as shown in Fig. 1, the original coefficient vector $a_l, l \in [0, N)$ is transformed into $a_{ijk}, i \in [0, n_1), j \in [0, n_2), k \in [0, n_3)$ in column-major order. Based on this transformation, the steps of the 3D NTT algorithm are outlined as follows [17]:

(1) Perform NTT along axis k on tensor A_0, i.e., $n_1 \times n_2$ n_3-point NTTs. Following we denote these NTTs as column-wise NTTs and the result tensor as A_1.

(2) Multiply each element in tensor A_1 with dimension-switching TFs $\omega_{n_2 \cdot n_3}^{k \cdot j}$ to obtain tensor A_2.

(3) Perform NTT along axis j on tensor A_2, i.e., $n_1 \times n_3$ n_2-point NTTs. Following we denote these NTTs as row-wise NTTs and the result tensor as A_3.

(4) Multiply each element in tensor A_3 with dimension-switching TFs $\omega_{n_1 \cdot n_2}^{j \cdot i} \times \omega_{n_1 \cdot n_2 \cdot n_3}^{k \cdot i}$ to obtain tensor A_4.

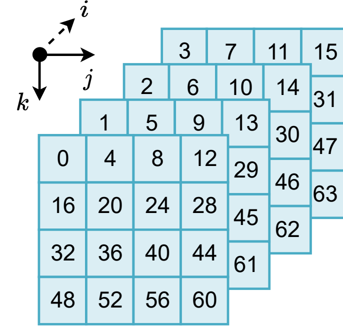

Fig. 1. An example of representing a polynomial coefficient vector of length 64 as a $4 \times 4 \times 4$ tensor.

(5) Perform NTT along axis i on tensor A_4, i.e., $n_2 \times n_3$ n_1-point NTTs. Following we denote these NTTs as depth-wise NTTs and the result tensor as A_5.

(6) Transform the tensor A_5 back to a one-dimensional vector in row-major order to obtain the final output.

For the INTT case, the algorithm steps are reversed, i.e., executed in the order of depth-wise INTTs $\rightarrow$ row-wise INTTs $\rightarrow$ column-wise INTTs, and required taking the inverse of TFs during dimension switching.

III. Hardware Architecture

A. Architecture Overview

The overall architecture of our proposed NTT/INTT is illustrated in Fig. 2, which takes n, m and $mode$ as inputs to dynamically support different parameter sets for NTT/INTT. n is set at compile time, while m can be changed at runtime and is a power of two less than or equal to n. We define $N = n \times n \times m$, fixing the polynomial degrees of the row-wise and the depth-wise NTTs/INTTs at n and dynamically configuring the polynomial degree of the column-wise NTTs/INTTs to achieve RTC. The architecture supports a dynamic configuration range from $n \times n \times 1$ to $n \times n \times n$, sufficiently covering most HE scenarios.

The architecture consists of four main components: a control unit (CU), coefficient memory (CM), PU and TFG. CU generates control and address signals, while CM and TFG provide

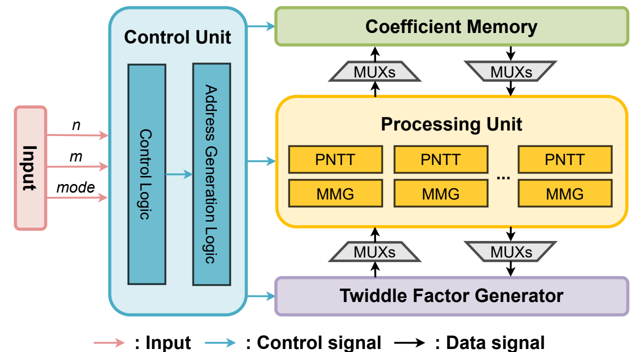

Fig. 2. Overall architecture of proposed NTT/INTT.

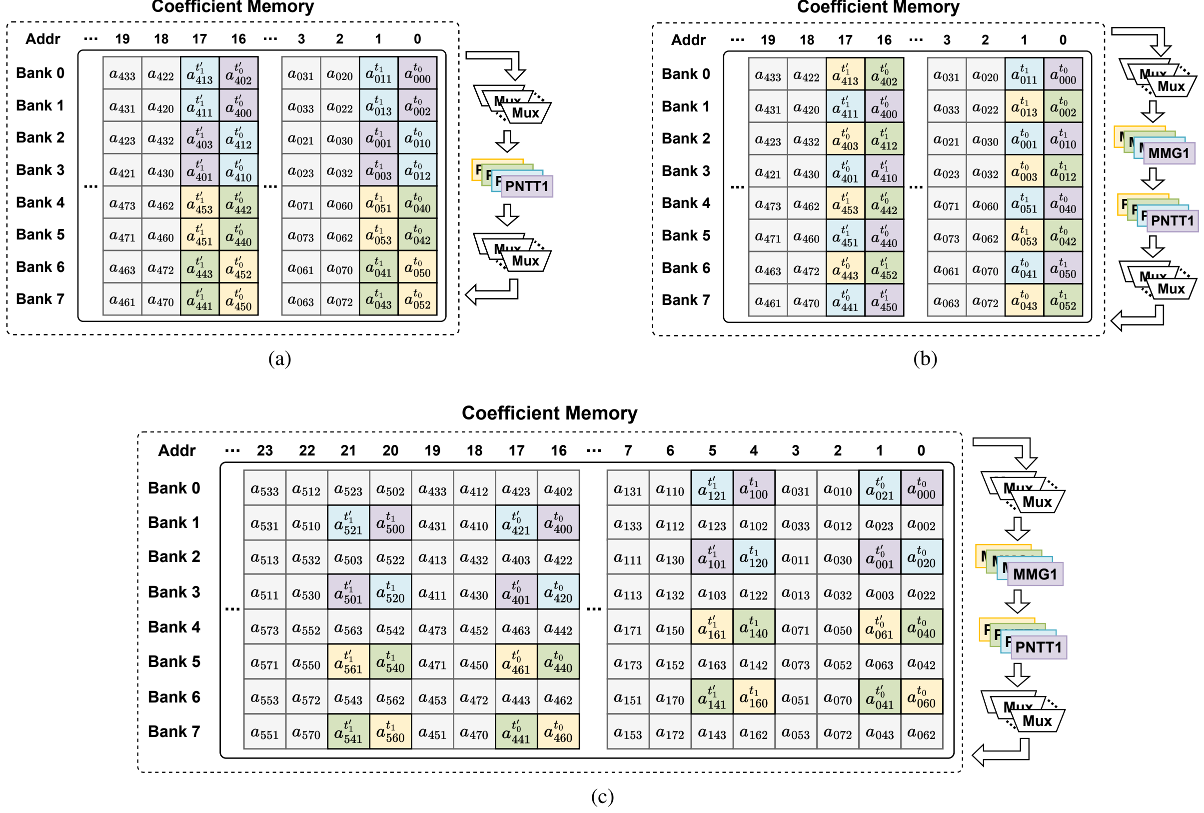

Fig. 3. Proposed memory access pattern for a 256-point NTT, which is further represented as a $8 \times 8 \times 4$ tensor. (a) The memory access pattern of the column-wise NTTs. (b) The memory access pattern of the row-wise NTTs. (c) The memory access pattern of the depth-wise NTTs.

polynomial coefficients and dimension-switching TFs to PU. PU comprises a series of PNTTs capable of supporting n-point or smaller NTT operations. Each PNTT is accompanied with a modular multiplier group (MMG) which contains two modular multipliers (MMs) for dot multiplication required during dimension switching and is reused for small-point NTTs within each dimension of 3D NTT. We select four PNTTs in this work to achieve sufficient parallelism and area efficiency. Notably, our proposed architecture is scalable to support higher parallelism.

Our PNTT is derived from the multi-path delay commutator (MDC) structure [10]. For an n-point NTT, the pipeline has $log_2 n$ PEs, each containing a butterfly unit (BFU), two FIFOs and two multiplexers. The FIFO depth varies across stages to achieve the desired access stride. Based on [21], our BFU supports both CT and GS butterfly for merged NTT/INTT. By dynamically changing the data paths between PEs, our PNTT enables run-time switching between NTT and INTT modes. Moreover, by selectively bypassing stages, our PNTT is capable of supporting polynomials with arbitrary power-of-two degrees $m (\leq n)$.

B. Conflict-free Memory Access Pattern

In this paper, we propose a conflict-free memory access pattern that avoids the need for extra hardware units by adjusting memory addresses and employing multiplexers for data exchange between CM and PU. We illustrate our memory access pattern using a 256-point NTT example, decomposed into an $8 \times 8 \times 4$ $(n = 8, m = 4)$ tensor. Each $k-j$ plane of the tensor is sequentially stored into CM by an ascending index i. Within each $k - j$ plane, columns $j = 0$ through $n/2 - 1$ are assigned to bank 0-3, while columns $j = n/2$ through $n - 1$ are allocated to bank 4-7. Coefficients from each column are interleaved across different banks to achieve the subsequent memory access pattern.

The memory access pattern is shown in Fig. 3. Each subplot shows two distinct planes, highlighting the first two clock cycles for each plane, denoted as t_0, t_1 for one plane and t'_0, t'_1 for the other. Specifically, Fig. 3a and Fig. 3b focus on planes with indices $i = 0, 4$, while Fig. 3c displays planes with indices $k = 0, 1$. The colors highlighted in CM correspond to the MMGs and PNTTs. During each clock cycle, the data is read out from CM and sent to the corresponding MMGs or PNTTs via multiplexers. Once the pipeline is filled, PNTTs write eight outputs back into CM following the same rule.

The memory access pattern for the column-/row-/depth-wise NTTs is different, mainly reflected in the memory addresses of each bank and the data switching performed by multiplexers. The column-wise and the row-wise NTTs operate on the $k-j$ planes indexed by i, following similar patterns across each plane. For instance, in the $i = 0$ plane depicted in Fig. 3a and

Fig. 3b, during the column-wise NTTs, eight banks initialize their addresses to 0 and increase by 1 each clock cycle. In the row-wise NTTs, for bank 0, 1, 4, 5, the initial value of the address is 0 and increases by $m/4$ per cycle, while bank 2, 3, 6, 7 starts at $m/4$ and alternates in subsequent cycles by first decreasing by $m/4$ and then increasing by $3m/4$, following a cyclic pattern. For the depth-wise NTTs on the $j-i$ planes indexed by k, consider the $k=0$ plane illustrated in Fig. 3c. Banks with even indices start their addresses from 0, while banks with odd indices begin at $n \times m/2$. Each clock cycle increases the address of all eight banks by $n \times m/8$. In this way, 3D NTT is performed naturally without extra hardware units.

The proposed memory access pattern covers all scenarios where $m \leq n$ and supports both NTT and INTT operations flexibly. Specifically, when $m = 1$, 3D NTT reduces to 2D, implemented by skipping the depth-wise NTT step. For the INTT operation, we execute the steps in reverse order. Based on this conflict-free memory access pattern and the PNTTs that support multiple polynomial degrees, we achieve a high-performance, area-efficient and dynamically configurable 3D NTT/INTT architecture.

C. On-the-Fly Twiddle Factor Generator

3D NTT introduces a substantial number of TFs required for dimension switching, limiting its application under the memory-bound constraint of HE. This paper proposes an on-the-fly TFG for dimension-switching TFs that aligns with the memory access pattern. In TFG, we allocate two $n \times n$ initial planes of TFs $\omega_{n \cdot n}^{l_1 \cdot l_2}$, an $n \times n$ buffer plane, and TF seeds $\omega_{n \cdot n \cdot m}^{l_1}$ for various m values, where $l_1, l_2 \in [0, n)$ and employ MMs for dynamic TF generation. As shown in Fig. 4, the two initial TF planes are stored in memory with different address ranges following a specific pattern. The l_2 direction of the initial planes is distributed across eight banks, while the l_1 direction is arranged in a sequence corresponding to the processing order of the row-wise NTTs $(0, 1, 4, 5, 2, 3, 6, 7,$ when $n = 8)$ and the depth-wise NTTs $(0, 2, 4, 6, 1, 3, 5, 7,$ when $n = 8)$. With increasing addresses, the TF values in the l_2 direction cyclically shift downwards by two positions of memory banks, exhibiting a periodic pattern every four consecutive addresses.

For $k-j$ dimension switching, the TFs are identical across different $k-j$ planes and depend on the run-time configurable parameter m, prepared in advance by squaring. In contrast, the TFs for $j-i$ dimension switching vary across different $j-i$ planes and are generated dynamically in real-time during computation. In the case of $k-j$ dimension switching, as shown in Fig. 4a, specific initial TF values $(l_1 = 0, 1, 2, 3,$ when $m = 4)$ are squared to obtain $\omega_{n \cdot m}^{k \cdot j}$ and stored cyclically in the buffer during the column-wise NTTs. During the row-wise NTTs, these precomputed TFs are sequentially fetched for PU following the cyclic pattern. The colored area in Fig. 4a illustrates the buffer outputs for the first clock cycle, with different colors indicating outputs to different MMGs. For $j-i$ dimension switching, as depicted in Fig. 4b, initial

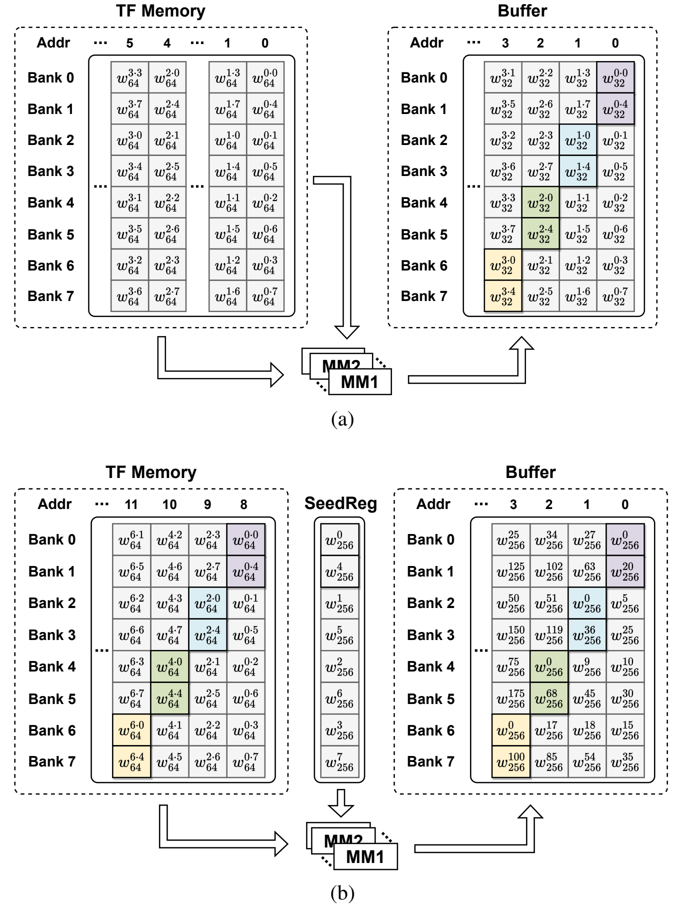

Fig. 4. On-the-fly TFG for 256-point NTT. (a) $k-j$ dimension switches during the column-wise and the row-wise NTTs. (b) $j-i$ dimension switches during the depth-wise NTTs.

TF values are cyclically output to PU and multiplied by the seeds $\omega_{n \cdot n \cdot m}^{i}$ to yield $\omega_{n \cdot n}^{j \cdot i} \times \omega_{n \cdot n \cdot m}^{i}$, which are then stored in the buffer. Subsequent $j-i$ TF planes are fetched from the buffer and the buffer is overwritten using the same process, allowing the buffer to be reused for generating all $j-i$ dimension-switching TFs $\omega_{n \cdot n}^{j \cdot i} \times \omega_{n \cdot n \cdot m}^{k \cdot i}$. With TFG, only minimal memory resources and eight MMs are required to dynamically generate all necessary dimension-switching TFs for various polynomial degrees.

The proposed on-the-fly TFG can also be applied to INTT. To avoid additional memory consumption for two extra initial TF planes required for INTT, we further propose a reuse mechanism. This mechanism obtains the necessary initial planes for INTT through a refresh operation. Specifically, the initial plane $\omega_{n \cdot n}^{-l_1 \cdot l_2}$ required for INTT satisfies:

$$\begin{aligned} \omega_{n \cdot n}^{-l_1 \cdot l_2} \bmod q &\equiv \omega_{n \cdot n}^{-l_1 \cdot l_2} \cdot \omega_{n \cdot n}^{n \cdot l_2} \cdot \omega_{n \cdot n}^{-n \cdot l_2} \cdot \omega_{n \cdot n}^{l_2} \cdot \omega_{n \cdot n}^{-l_2} \bmod q \\ &\equiv \omega_{n \cdot n}^{(n-l_1-1) \cdot l_2} \cdot \omega_{n \cdot n}^{-n \cdot l_2} \cdot \omega_{n \cdot n}^{l_2} \bmod q \qquad (1) \end{aligned}$$

According to (1), the values of $\omega_{n \cdot n}^{l_1 \cdot l_2}$ and $\omega_{n \cdot n}^{(n-l_1-1) \cdot l_2}$ are identical, differing only in their memory address order. Based on this observation, we can multiply the original initial

TABLE I
IMPLEMENTATION RESULTS OF NTT/INTT ARCHITECTURE AND COMPARISON TO PRIOR WORKS

Work	**Dev.**[a]	log_2N /	**Resource**				**Freq.**	**Latency**		**Throughput**	**EqS**[b]	**TPS**[c]
		log_2q	LUT	FF	DSP	BRAM	(MHz)	(CCs)	(μs)	(Mbps)	(K)	
Mert [5]	V690T	12 / 60	99.4K	-	992	176	125	972	7.70	31917	167.33	0.19
Ye [9]	V485T	12 / 60	17.0K	11.0K	286	24.5	150	8284	55.23	4450	39.23	0.11
Su [6]	VU190	14 / 60	99.4K	93.2K	1080	3680	300	28672	95.57	10286	495.34	0.02
Öztürk [7]	V690T	14 / 32	219.2K	90.8K	768	193	250	-	24.50	21400	178.30	0.12
		15 / 32							50.90	20601		0.12
Roy [8]	V240T	16 / 30	72.6K	63.1K	250	84	100	47795	477.95	4114	63.27	0.07
Hirner [10]	V485T	16 / 64	37.5K	29.9K	320	383	135	65706	486.71	8618	131.15	0.07
Kim [11]	VU190	17 / 62	365.0K	335.0K	1332	2258	200	98304	491.52	16533	376.20	0.04
Ours	ZU102	12 / 60	75.2K	42.6K	428	508	259	1197	4.62	53195	90.34	0.59
		13 / 60						3260	12.59	39041		0.43
		14 / 60						6340	24.48	40157		0.44
		15 / 60						12493	48.24	40756		0.45
		16 / 60						24792	95.72	41080		0.45
		17 / 60						49383	190.67	41246		0.46
		18 / 60						98558	380.53	41334		0.46

[a] Device: V690T: Virtex-7 XC7VX690T; V485T: Virtex-7 XCVX485T; VU190: Virtex UltraScale XCVU190; V240T: Virtex-6 XC6VLX240T-1FF1156; ZU102: Zynq UltraScale+ ZCU102.
[b] Equivalent slice (EqS) = #Slice + Eq. DSP + Eq. BRAM. #Slice $\approx \frac{\#LUT}{4}$ (7 Series) / $\frac{\#LUT}{8}$ (UltraScale), 1 DSP $\approx$ 102.4 Slices (7 Series) / 51.2 Slices (UltraScale), 1 BRAM $\approx$ 232.4 Slices (7 Series) / 116.2 Slices (UltraScale) [22].
[c] Throughput per slice (TPS) [22].

planes by the refresh factor $\omega_{n\cdot n}^{-n\cdot l_2} \cdot \omega_{n\cdot n}^{l_2}$ and read the TFs in reverse order to achieve initial planes reuse between NTT and INTT. This approach further reduces the memory overhead for dimension-switching TFs. It should be noted that since NTT and INTT are generally not performed consecutively in HE, the refresh time can be hidden without affecting performance.

IV. Results and Discussion

A. Experimental Setup

Our design takes the maximum degree n supported by PNTTs and the modulus bit width log_2q as compile-time inputs to generate synthesizable Verilog HDL code. We dynamically configure the polynomial degree by adjusting the value of m at runtime. The Xilinx Vivado Design Suite is used for synthesis and place & route stages on the Xilinx Zynq UltraScale+ ZCU102 FPGA platform. To evaluate the area and performance of our design under HE parameters, we set $log_2q = 60$ and $n = 64$, enabling the accelerator to support polynomial degree ranging from 2^{12} to 2^{18}, covering most HE scenarios. Latency is determined by the ratio of clock cycles (CCs) to the clock frequency, while throughput refers to the number of bits processed by the accelerator per unit of time, calculated as follow:

$$\text{Throughput } (Mbps) = \frac{\text{Number of bits } (bits)}{\text{Latency } (\mu s)} \tag{2}$$

B. Comparison and Discussion

We compare our design with prior works on FPGA [5]–[11], as shown in Table I. Due to different hardware platforms and parameter sets between different studies, we use equivalent slice (EqS) to measure resource consumption and throughput per slice (TPS) as a comparative metric for a fair comparison [22]. EqS combines LUTs, FFs, DSPs, and BRAMs into a unified measure, defined as follows,

$$\text{EqS} = \text{\#Slice} + \text{Eq. DSP} + \text{Eq. BRAM} \tag{3}$$

where #Slice approximately equals to #LUT$/4$ for 7 series FPGAs and #LUT$/8$ for UltraScale FPGAs. One DSP block and one BRAM unit can be replaced by 102.4, 232.4 and 51.2, 116.2 slices for 7 series and UltraScale FPGAs, respectively.

TPS serves as a measure of the area efficiency, defined as follows:

$$\text{TPS } (Mbps/Slice) = \frac{\text{Throughput } (Mbps)}{\text{EqS } (Slice)} \tag{4}$$

Several studies implemented HE parameter sets [7], [8], [10], [11]. Öztürk et al. [7] designed a large-degree NTT-based polynomial multiplier for somewhat homomorphic encryption (SWHE) based schemes, utilizing a large number of multipliers and on-chip memory, leading to suboptimal area efficiency. Our design, with its optimized memory access pattern and TFG, achieves better performance and TPS. Roy et al. [8] proposed a single-FPGA design of FV and optimized the memory access to parallel several cores for the butterfly operations. However, their operating frequency is relatively low. Our design, by simplifying the data dependencies between NTT stages of PU, achieves a higher frequency, leading to a 9.9× improvement in throughput and a 6.4× improvement in TPS. Hirner et al. [10] proposed PROTEUS, an open-source parameterized tool for generating synthesizable pipelined NTT architectures. While their MDC architecture is similar to our PNTTs, our use of the 3D NTT algorithm and the simplified PU achieves a 4.8× improvement in throughput and a

6.4× improvement in TPS. Kim et al. [11] proposed a fully pipelined INTT architecture supporting a polynomial degree $N = 2^{17}$. However, their design requires five intermediate buffers for coefficient reordering to avoid pipeline stall, leading to increased memory consumption. Our conflict-free memory access pattern can implement the 3D NTT algorithm without the need for extra hardware units, resulting in a 11.5× improvement in TPS. Furthermore, although our computing and memory resources are sufficient to support NTT up to 2^{18} polynomial degrees, our design maintains an advantage of performance and TPS even with small parameter sets [5], [6], [9].

V. Conclusion

This paper presents an FPGA implementation of an RTC NTT/INTT accelerator supporting HE parameter sets based on the 3D NTT algorithm. To mitigate the extra overhead induced by 3D NTT, we propose a conflict-free memory access pattern and an on-the-fly TFG. Compared to state-of-the-art works, our design achieves significant improvements in performance and area efficiency.

References

[1] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, "Sok: Fully homomorphic encryption accelerators," *arXiv preprint arXiv:2212.01713*, 2022.

[2] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 2019, pp. 347–368.

[3] A. C. Mert, E. Öztürk, and E. Savaş, "Fpga implementation of a run-time configurable ntt-based polynomial multiplication hardware," *Microprocessors and Microsystems*, vol. 78, p. 103219, 2020.

[4] X. Hu, J. Tian, M. Li, and Z. Wang, "Ac-pm: An area-efficient and configurable polynomial multiplier for lattice based cryptography," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, pp. 719–732, 2023.

[5] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2829–2843, 2022.

[6] 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.

[7] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, pp. 3–16, 2017.

[8] 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.

[9] Z. Ye, R. C. Cheung, and K. Huang, "Pipentt: A pipelined number theoretic transform architecture," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 10, pp. 4068–4072, 2022.

[10] F. Hirner, A. C. Mert, and S. S. Roy, "Proteus: A tool to generate pipelined number theoretic transform architectures for fhe and zkp applications," *Cryptology ePrint Archive*, 2023.

[11] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 56–64.

[12] 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 36*. Springer, 2021, pp. 98–117.

[13] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 416–428.

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

[15] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[16] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th annual international symposium on computer architecture*, 2022, pp. 711–725.

[17] 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)*. IEEE, 2023, pp. 1–9.

[18] X. Chen, W. Lu, T. Su, and D. Chen, "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)*. IEEE, 2024, pp. 1–5.

[19] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology–LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings 2*. Springer, 2012, pp. 139–158.

[20] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers," in *International conference on cryptology and information security in Latin America*. Springer, 2015, pp. 346–365.

[21] 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.

[22] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, pp. 2459–2463, 2019.