An Efficient and Scalable Hardware Architecture for Number Theoretic Transform on FPGA with Design Automation

Yilan Zhu[1*], Geng Yang[1*], Xingyu Tian[2], Dilshan Kumarathunga[2], Liang Kong[1],
Xianglong Deng[3], Shengyu Fan[3], Guang Fan[1], Guiming Shi[4], Lei Chen[1],
Bo Zhang[1], Yisong Chang[1], Shoumeng Yan[1], Zhenman Fang[2] and Mingzhe Zhang[1†]
[1] Ant Group, Hangzhou, China. [2] Simon Fraser University, Burnaby, BC, Canada.
[3] Institute of Information Engineering, CAS, Beijing, China. [4] Tsinghua University, Beijing, China.
yilanzhu02@163.com, yanggeng.yang@antgroup.com, smartzmz@gmail.com

***Abstract*—Fully Homomorphic Encryption (FHE) has become a promising approach to protecting data privacy in emerging application scenarios. Unfortunately, FHE suffers from significant processing speed degradation compared to plaintext computation, with one of the primary bottlenecks being the time-consuming Number Theoretic Transform (NTT). Therefore, accelerating NTT to accommodate various FHE parameters is crucial to advancing FHE towards practical use. With highly reconfigurable and performant logical fabrics, Field Programmable Gate Arrays (FPGAs) have exhibited great potential in NTT acceleration.**

By decomposing large-point NTT with strong data dependency into independent and simple small-point NTTs, the emerging Ten-step NTT (TNTT) algorithms intuitively enable higher parallelism and thereby have the potential to explore better performance compared to traditional algorithms. However, our quantitative analysis reveals that TNTT exhibits significant performance degradation as parallelism increases due to additional varying-size transpositions and Hadamard products.

This paper proposes AutoNest, an efficient and scalable hardware architecture, along with an accelerator auto-generation framework for TNTT. The proposed hardware architecture maximizes performance by 1) adopting a 2D block decomposition dataflow to address critical path delays in transpose logic, thereby improving clock frequency. 2) integrating algorithm-level cost-free twiddle factor fusion to reduce the number of modular multiplications in Hadamard products, thereby allowing higher parallelism on chip. Moreover, we also deliver an accelerator generation framework conducting automated design space exploration to elaborate a performant TNTT architecture under the target FPGAs' resource budget for user-defined FHE parameters. Experimental results on the AMD-Xilinx U280 FPGA demonstrate that NTT accelerators generated by AutoNest achieve an average speedup of 2.31× compared to prior designs.

I. Introduction

Driven by the rapidly growing demand for secure systems in the field of machine learning, Fully Homomorphic Encryption (FHE) has attracted significant attention for its ability to perform computations on encrypted data without requiring decryption [13], [16]. However, existing lattice-based FHE schemes, such as BFV [7], TFHE [12], and CKKS [10], face computational overheads several orders of magnitude higher than plaintext operations, posing challenges for practical deployment. The Number Theoretic Transform (NTT) is an essential operation in FHE, as it significantly reduces the computational complexity of polynomial multiplications from $O(N^2)$ to $O(N \log N)$ for N-sized polynomials. According to [16], [29], NTT accounts for more than 50% of the total execution time for different FHE applications. Therefore, accelerating NTT becomes critically important to improving the overall efficiency of FHE implementations.

In recent years, there have been extensive studies on accelerating NTT by leveraging different hardware platforms, such as GPUs [16], FPGAs [3], and ASICs [13], to improve both the speed and efficiency of NTT computations. Due to their reconfigurability, FPGAs are a prominent choice for accelerating non-linear operators such as NTT [18], [24], [31], [35]. However, long data dependencies in NTT, especially for a large degree of polynomial in FHE, result in complex data reorganization logic and significantly impact the performance of FPGA implementations.

Fortunately, multi-step NTT algorithm [30] adopts multi-dimensional decomposition, which partitions the original N-point NTT into multiple independent and smaller n-point NTTs ($n \ll N$), significantly reducing data dependencies. Benefiting from these small-point NTTs, multi-step NTT intuitively enables higher parallelism implementation, thereby achieving better performance on FPGAs. This paper focuses on recent Ten-Step NTT (TNTT) [28] based on four-dimensional decomposition. However, our quantitative analysis reveals that the TNTT implementation performs poorly with increasing parallelism due to additional varying-sized transpositions and Hadamard products.

Specifically, TNTT relies on the transposition to convert between the row-major and column-major orders of matrix-structured polynomials. However, as the degree of parallelism grows, the complexity of the data reorganization logic required for transposition increases significantly, resulting in longer critical path delays and thus a sharp decline in clock frequency. Experimental results reveal that when data parallelism increases to 128 for processing polynomials with degree of 2^{12}, TNTT implementation even faces placing and routing failures on the real AMD-Xilinx U280 FPGA. In addition, the varying-sized transpositions interleaved between different steps of TNTT present increasing challenges, as they must be meticulously optimized to ensure alignment with the pipeline efficiency of the subsequent small-point NTT computations.

An additional operation in TNTT is the Hadamard product based on modular multiplication (mod-mult) operations. Experimental results reveal that the Hadamard product contributes significantly to resource utilization, with DSP con-

* Both author contributed equally to this research.
† Mingzhe Zhang is the corresponding author.

sumption accounting for approximately 27%∼33% of the total under different parallelism and polynomial degrees. Therefore, as the parallelism expands, the resource demands of the TNTT accelerator increase more rapidly compared to the original NTT. This poses significant challenges to achieving performance improvements on resource-constrained FPGAs.

To address these challenges, we propose AutoNest, an efficient and scalable hardware architecture together with an accelerator auto-generation framework for TNTT. Proposed fully-pipelined architecture adopts a dedicated dataflow based on 2D block decomposition to address critical path delays in transpose logic while seamlessly adapting smaller-point NTT computations of different steps in TNTT. Experimental results demonstrate that the proposed dataflow effectively partitions large-sized transpositions into parallel block-level transpositions under high parallelism, leading to an improvement of 39∼137MHz in clock frequency. Furthermore, we also propose algorithm-level fusion that utilizes offline computation and reorder of twiddle factors to reduce the number of mod-mults in Hadamard products. Building on this, we propose a corresponding hardware implementation that is cost-free and adaptable to different parallelism settings. Experimental results show that the proposed optimized design reduces DSP usage by 13.6%∼16.7% for different FHE parameters.

Based on the proposed scalable architecture through algorithm-hardware co-design, we deliver an accelerator generation framework conducting automated design space exploration, instantiation of configurable HLS (high-level synthesis)-described units, and FPGA design flow to elaborate a performant TNTT architecture under user-defined NTT parameters. Evaluated on an AMD-Xilinx U280 FPGA, AutoNest achieves a 2.31× speedup over prior works across different polynomial degrees N. For the parameter configuration $N = 2^{12}$ and bit-width $\log q = 32$, AutoNest achieves an average latency of 0.16 us, delivering a 2.69× average improvement over existing state-of-the-art (SOTA) FPGA designs. Furthermore, we further demonstrate the effectiveness of AutoNest in supporting various multi-step NTT variants, and evaluate its performance advantages within complete FHE applications.

In summary, this paper makes the following contributions:

1. We propose a novel dataflow strategy based on 2D block decomposition, which facilitates the efficient execution of block-wise transposition with varying sizes in the TNTT. This method not only alleviates critical path delays for better frequency performance but also ensures seamless adaptation with the following small-point NTT computations, thereby maximizing pipeline efficiency.
2. We introduce an algorithm-hardware co-optimization for Hadamard products that features an algorithm-level twiddle factor fusion to reduce the number of modular multipliers, and a corresponding hardware-level, cost-free design that is adaptable to various parallelism configurations.
3. Building on proposed scalable, fully pipelined architecture through algorithm-hardware co-design, we develop a complete automated framework for system-level design space exploration. The framework enables the determination of optimal parameter configuration while further automating the generation of HLS-based code for efficient and seamless hardware implementation.

TABLE I. Parameters and Notations in NTT.

Variable	Explanation
N	Polynomial degree
n_{ij}	NTT length in step 1/4/7/10 for TNTT
q	the prime modulus
$\log q$	the bitwidth of prime modulus
ψ	the primitive $2N$-th root of unity in Z_q

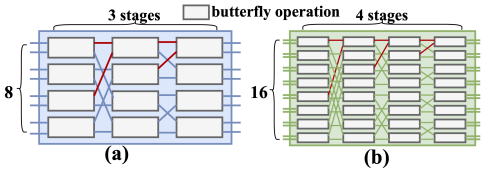


Fig. 1. NTT Example including (a) NTT_8: $N = 8$ input data with 3 butterfly stages and (b) NTT_{16}: $N = 16$ input data with 4 butterfly stages.

II. Background

A. Fundamentals of NTT in FHE

NTT optimizes polynomial multiplication by reducing its complexity from $O(N^2)$ to $O(N \log N)$, which is widely applied in FHE. Fig.1 shows the computation process of Standard NTT (SNTT) for polynomial sizes $N = 8$ and $N = 16$. Similar to the Fast Fourier Transform, it uses a divide-and-conquer strategy to break down the evaluation of an N-degree polynomial into $\log N$ stages, with $N/2$ butterfly operations in each stage. Butterfly operation is the core of the NTT computation, consisting of modular operations performed under modulus q. A widely used Cooley-Tukey butterfly is computed as: $x_{out} = x_{in} + y_{in} \cdot \psi^k, \quad y_{out} = x_{in} - y_{in} \cdot \psi^k$. ψ is a primitive $2N$-th root of unity exists in Z_q. $\psi^k, k \in [0, 2N)$ are precomputed values called the twiddle factors (TF).

In different FHE schemes such as CKKS [21], BFV [15], and TFHE [11], polynomials in Z_Q are typically represented as long vectors, with N ranging from 2^{12} to 2^{16}, and coefficients that can be up to thousands of bits. To reduce the computational complexity of using large integer coefficients in NTT, the Residue Number System (RNS) based on Chinese Remainder Theorem is widely utilized [9]. Using RNS, a large-bitwidth polynomial in Z_Q is decomposed into L smaller-bitwidth polynomial limbs in Z_{q_i} $(0 < i \leq L)$, where the primes q_i typically have a bit-width $(\log q_i)$ of ranging from 32 to 64 bits. Due to the numerous limb-based NTT in FHE, optimizing the throughput of NTT is a primary objective.

B. Multi-Step NTT Algorithm

For NTT in FHE, the large polynomial N exacerbates data dependencies between butterfly operations across different NTT stages, as highlighted in the red line of Fig.1, thus hindering highly efficient execution. To mitigate this issue, multi-step decomposition algorithms were introduced [4], [30]. The core concept is to map a large, one-dimensional NTT onto a multi-dimensional structure (such as three-, four-, or five-dimensional), thereby decomposing it into multiple

Algorithm 1: Ten-step NTT (TNTT) Algorithm

Input: Polynomial $a(x) \in \mathbb{Z}_q[x]/(x^N+1)$; Primitive $2N$-th root of unity $\psi \in \mathbb{Z}_q$;
$N = n_1 \cdot n_2 = n_{11} \cdot n_{12} \cdot n_{21} \cdot n_{22}$
Output: $\bar{a}(x) = \mathrm{NTT}(a) \in \mathbb{Z}_q[x]/(x^n+1)$

for $i = 0$ **to** $n_{12} \cdot n_{21} \cdot n_{22} - 1$ **do**
  $d[i] \leftarrow \mathrm{NTT}_{n_{11}}(a[i], \psi_{n_{11}})$; ▷ Step 1
$d \leftarrow \mathrm{d}_{\mathrm{n_{11}\times n_{12}}} \odot \psi_{\mathrm{n_{11}n_{12}}}$; ▷ Step 2
$d \leftarrow \mathrm{transpose}_{\mathrm{n_{11}\times n_{12}}}(d)$; ▷ Step 3
for $i = 0$ **to** $n_{11} \cdot n_{21} \cdot n_{22} - 1$ **do**
  $c[i] \leftarrow \mathrm{NTT}_{n_{12}}(d[i], \psi_{n_{11}})$; ▷ Step 4
$c \leftarrow \mathrm{d}_{\mathrm{n_1\times n_2}} \odot \psi_{\mathrm{n_1n_2}}$; ▷ Step 5
$c \leftarrow \mathrm{transpose}_{\mathrm{n_1\times n_2}}(c)$; ▷ Step 6
for $i = 0$ **to** $n_{11} \cdot n_{12} \cdot n_{22} - 1$ **do**
  $b[i] \leftarrow \mathrm{NTT}_{n_{21}}(c[i], \psi_{n_{21}})$; ▷ Step 7
$b \leftarrow \mathrm{b}_{\mathrm{n_{21}\times n_{22}}} \odot \psi_{\mathrm{n_{21}n_{22}}}$; ▷ Step 8
$b \leftarrow \mathrm{transpose}_{\mathrm{n_{21}\times n_{22}}}(b)$; ▷ Step 9
for $i = 0$ **to** $n_{11} \cdot n_{12} \cdot n_{21} - 1$ **do**
  $a[i] \leftarrow \mathrm{NTT}_{n_{22}}(b[i], \psi_{n_{22}})$; ▷ step 10

smaller, independent NTTs. Specifically, the recent ten-step NTT (TNTT) as illustrated in Alg.1 divides a one-dimensional N-point NTT into a four-dimensional structure of smaller n_{ij}-point NTTs, where N is factored as $N = n_{11} \cdot n_{12} \cdot n_{21} \cdot n_{22}$. Compared to SNTT, multi-step NTT intuitively offers the hardware-friendly advantage as it facilitates the parallel execution of a large number of independent small-point NTTs ($n \ll N$).

However, this NTT decomposition inevitably introduces varying-sized transpositions and mod-mult-based Hadamard products. Specifically, as shown in Alg.1 of TNTT, the Hadamard product performs element-wise mod-mults on two matrix-structured polynomials, while matrix transposition swaps rows and columns of the matrix. These operations bring additional cost and require efficient implementation. This paper focuses on accelerating TNTT under a four-dimensional structure and further extends our hardware architecture to support other dimensional configurations. We list the parameters of TNTT involved in this paper as shown in Table I.

C. FPGA-based NTT Acceleration

With highly reconfigurable and performant logical fabrics, FPGAs offer an opportunity to balance performance and flexibility. Compared to GPUs, FPGA provides a distinct advantage in performing nonlinear modular arithmetic efficiently. In contrast to ASICs, FPGA offers greater flexibility by allowing efficient reprogramming for NTT implementations with varying parameter values. Therefore, FPGA holds significant potential for accelerating NTT.

The primary goal in designing a high-throughput FPGA-based NTT accelerator is to maximize its computational throughput. Fundamentally, throughput is determined by the interplay between the clock frequency and the number of clock cycles required per operation. This relationship can be expressed as:

$$\text{Throughput} = \frac{\text{Frequency}}{\text{Cycles per Operation}} \tag{1}$$

Crucially, both terms in this equation are functions of the number of parallel Butterfly Units (BUs) that perform the primary computation of NTT.

On one hand, the number of cycles per operation is inversely proportional to the number of BUs. For a given polynomial degree, the total workload of butterfly operations is fixed. Therefore, instantiating more BUs within limited FPGA resources to process data in parallel directly reduces the number of clock cycles needed to complete an entire NTT computation, which serves to increase throughput.

On the other hand, the achievable clock frequency is constrained by the critical path delay. As we increase the number of BUs, the dense computational logic and complex interconnections lead to greater routing complexity on the FPGA fabric. This, in turn, exacerbates the critical path delay and consequently limits the maximum achievable clock frequency, creating a counteracting effect on throughput. Therefore, the optimal design lies in finding a balance that maximizes the product of data parallelism and the resulting frequency. This paper investigates this performance bottleneck by focusing on these two keys, interdependent aspects.

III. Motivation

In this section, we provide a quantitative analysis of TNTT performance to explore the opportunities and challenges in accelerator design.

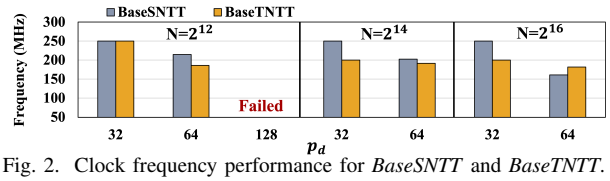


Fig. 2. Clock frequency performance for *BaseSNTT* and *BaseTNTT*.

A. Trade-off between Hardware Utilization and Frequency

We first evaluate the performance of TNTT on the real AMD-Xilinx U280 FPGA. To ensure representativeness, baseline SNTT architecture (called *BaseSNTT*) adopts SOTA dataflow architecture from open-sourced AutoNTT [24]. Baseline TNTT architecture (called *BaseTNTT*) extends AutoNTT's architecture by incorporating additional transpose units [22] for steps 3/6/9 and group mod-mult units for Hardmard product in steps 2/5/8 of Alg.1. The data bit-width ($\log q$) adopts the widely used 32-bit in FHE [22]. The implementation of BU utilizes optimized word-level Montgomery reduction [26]. We denote data parallelism by p_d, indicating the number of polynomial data processed in parallel per cycle.

As shown in Fig.2, the clock frequency of *BaseTNTT* falls short of expectations, offering no performance advantage over *BaseSNTT* at the same level of parallelism, despite employing hardware-friendly NTT decomposition. It is evident that the NTT decomposition simplifies the data reorganization between NTT stages. However, extra Hadamard products and matrix

Algorithm 2: Pseudocode of TPU for $n_1 \times n_2$ tranposistion of step 6 in TNTT.

Input: num input N-Polynomial datastream with p_d data parallelism: I ;
Output: num output N-Polynomial datastream with p_d data parallelism: O ;

1 $BufA[\frac{N}{p_d}][\frac{N}{p_d}][p_d][p_d]$;
2 `#Pragma HLS ARRAY PARTITION dim=4`
3 $BufB[\frac{N}{p_d}][\frac{N}{p_d}][p_d][p_d]$;
4 `#Pragma HLS ARRAY PARTITION dim=4`
5 **for** $m = 0$ **to** num **do**
6 **for** $x_1 = 0$ **to** $\frac{N}{p_d}$ **do**
7 **for** $x_0 = 0$ **to** p_d **do**
8 **for** $y_1 = 0$ **to** $\frac{N}{p_d}$ **do**
9 `#Pragma HLS PIPELINE`
10 **for** $y_0 = 0$ **to** p_d **do**
11 `#Pragma HLS UNROLL`
12 $v = (y_0 + x_0)\ \%\ p_d$;
13 **if** $m\%2 == 0$ **then**
14 $BufA[x_1][y_1][x_0][v] \leftarrow I[y_0]$;
15 $O[y_0] \leftarrow BufB[y_1][x_1][y_0][v]$;
16 **else**
17 $BufB[x_1][y_1][x_0][v] \leftarrow I[y_0]$;
18 $O[y_0] \leftarrow BufA[y_1][x_1][y_0][v]$;

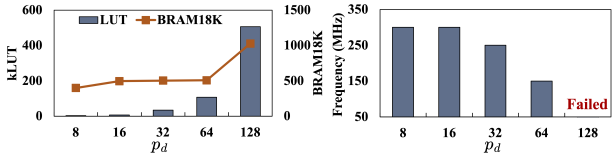


Fig. 3. The impact of transpose with different data parallelism p_d on resources consumption and clock frequency when $N = 2^{16}$.

transpositions come at a cost. Therefore, we focus on thoroughly analyzing the impact of these two operations on the overall TNTT performance in the following subsections.

B. Impact of Transposition on Frequency

As shown in Alg.1, the transposition operation, which swaps matrix rows and columns, occurs multiple times (step 3/6/9) in TNTT for varying matrix sizes. A representative efficient implementation in *BaseTNTT* and existing work [8], [22] involves storing input matrices row-wise (or column-wise) into the diagonal positions of a matrix buffer and subsequently reading them out in the pattern of transposed rows (or columns). Alg.2 shows the computation process, where the data parallelism p_d affects performance. We observe that both *BaseTNTT* and existing works [8], [22], [33] typically limit p_d to 32 and do not show higher parallelism. This level of parallelism underutilizes FPGA resources, with utilization falling below 30% both in *BaseTNTT* and [8], [22], [33].

To assess the impact of the transpositions on TNTT performance, we evaluate parallelism p_d from 8 to 128 for transposition in step 6 as an example shown in Fig.3. At lower p_d (e.g., 8 or 16), a clock frequency of 300 MHz is easily achieved. However, as data parallelism p_d increases, it not only significantly raises the hardware costs, as shown in Fig.3, but also leads to a dramatic drop in clock frequency. When the parallelism increases to 128, routing failures are encountered. To identify the cause, we analyze Alg. 2 and find

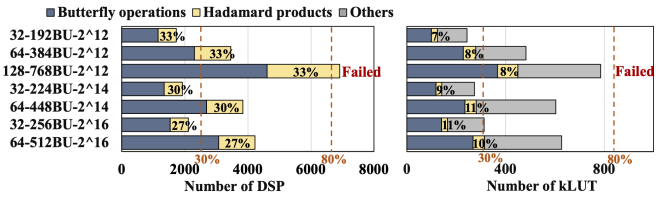


Fig. 4. Resource breakdown of *BaseTNTT* with different data parallelism p_d for different N. The orange vertical lines in the figure represent 30% or 80% of the total resource of the U280 FPGA.

that p_d parallelism divides the matrix-structured polynomial into several $p_d \times p_d$ transpose blocks in lines 1–4. As p_d increases, increasing the size of transpose blocks intensifies memory access complexity, which exacerbates critical path delays, ultimately leading to a continuous decline in clock frequency. Consequently, when striving to maximize the data parallelism of TNTT for optimal throughput, additional transposition emerges as a pivotal focus of design optimization.

C. Impact of Hadamard Products on Hardware Overhead

The Hadamard product in TNTT is derived from the decomposition of the original twiddle factors and employs the same mod-mult computation as the butterfly operation. To analyze the impact of additional Hadamard products on area utilization, a resource breakdown of the *BaseTNTT* is conducted. Fig.4 illustrates the resource utilization percentages of core butterfly operations and extra Hadamard product operations across data parallelism p_d for different N. It can be observed that the Hadamard product consumes a significant number of DSPs, especially when $N = 2^{12}$, where the DSP usage for the Hadamard product accounts for 33% of the total DSPs. When $N = 2^{16}$, the Hadamard product still occupies 27% of the total DSPs. In terms of LUT consumption, the Hadamard product does not dominate as strongly as it does in DSP usage, but it nonetheless accounts for approximately 7% to 11% of the total LUTs. When $p_d = 128$ for $N = 2^{12}$, *BaseTNTT* with 768 BUs utilizes around 80% of both DSPs and LUTs on the U280 FPGA, leading to placement and routing failures.

The experiment demonstrates that the Hadamard product consumes significant on-chip computation resources, limiting the deployment of more BUs for higher parallelism and ultimately reducing the performance of the TNTT accelerator.

D. Opportunities and Challenges

Through comprehensive quantitative analysis, we identify opportunities and challenges for extra transpositions and Hadamard products that must be addressed to fully exploit the performance potential of TNTT on FPGAs.

1) Extra Transposition: Quantitative analysis in Sec.III-B indicates that increasing parallelism p_d leads to a larger transposition block of $p_d \times p_d$ shown in Alg.2, which in turn causes a noticeable degradation in clock frequency. A widely adopted method is appropriate parallel tiling for high-parallelism transpositions, as it provides an opportunity to mitigate critical path delay and help maintain desirable clock frequencies. However, implementing this approach for TNTT is non-trival because **(i)**: The tiling strategy must balance

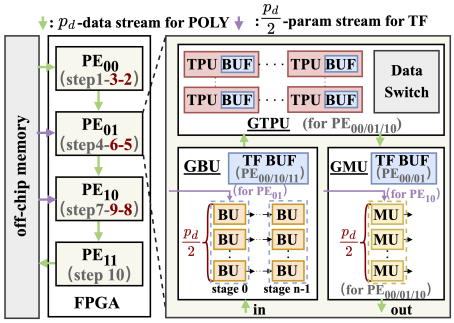


Fig. 5. Overall hardware architecture for AutoNest.

the data streams across varying-sized transpositions (e.g., $n_{11} \times n_{12}$ in step 3, $n_1 \times n_2$ in step 6, and $n_{21} \times n_{22}$ in step 9) while maintaining the pipeline efficiency of the following $\text{NTT}_{n_{ij}}$ computation. **(ii)**: A trade-off exists between tile size and the number of tiles, requiring exploration to identify the optimal tiling solution.

2) Extra Hadamard Products: Results from the performance evaluations in Sec.III-C highlight that the high resource utilization of mod-mults (DSPs and LUTs) in the Hadamard product significantly limits the achievable parallelism, leading to performance degradation of *BaseTNTT*. From the algorithmic analysis, we find that cascaded mod-mult operations in both NTT and Hadamard products use constant twiddle factors, enabling an offline twiddle factor fusion opportunity to reduce online computation. However, frequent transpositions between small-point NTTs and Hadamard products reorder data elements, thus disrupting data flow. It poses a challenge in reducing the overhead of the Hadamard product.

Building upon the opportunities offered by TNTT, this paper proposes a high-performance, scalable FPGA-based dataflow hardware architecture in Sec.IV to address the challenges of additional transpositions and Hadamard products, while Sec.V presents a system-level automated design space exploration tool for identifying the optimal configurations and HLS-based code generation across diverse FHE parameters.

IV. AutoNest

A. Architecture Overview

Fig.5 presents our efficient and scalable TNTT architecture with algorithm-level co-optimization. To reduce the overhead of off-chip memory accesses, we independently instantiate processing units (PEs) for each of the 10 steps of TNTT. PE_{00}, PE_{01}, PE_{10}, and PE_{11} are connected in a pipeline, forming a coarse-grained dataflow structure. Each PE maintains a consistent data rate, processing p_d data in parallel. Each PE_{ij} contains a group of BUs, transpose units (TPU), and hardamard product units (MU) with a fine-grained dataflow, which are referred to as GBU, GTPU, and GMU, respectively.

To address frequency degradation caused by large-sized transpositions under high data parallelism, we propose fine-grained dataflow based on 2D block decomposition strategy. This approach enables smaller block-wise transpositions within GTPU while ensuring the correct execution of $NTT_{n_{ij}}$ in GBU with negligible latency overhead. To reduce the additional resource overhead of mod-mults caused by extra

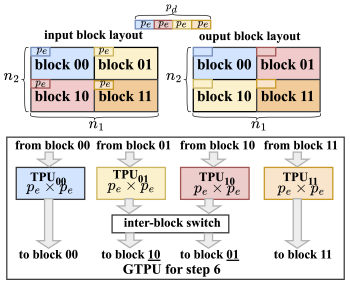


Fig. 6. Dataflow execution of GTPU in PE_{01} with an example of setting $p_b = 4, p_e = 4$ when $p_d = 16, N = 256, n_1 = n_2 = 16, n_{ij} = 4$.

Hadamard products, we proposed a cost-free twiddle factor fusion algorithm and develop the resource-efficient GMU design, which reduces the number of mod-mult from p_d to $\frac{p_d}{2}$ for each GMU. Note that the execution order of the Hadamard product step and transposition step must be reversed to ensure the correctness of the optimization algorithm.

Based on proposed FPGA core, the polynomial datastream with p_d parallelism is pipelined from off-chip memory to on-chip for executing TNTT computation, and the final results are streamed back to the off-chip memory. The TF parameters for the last stage of GBU in PE_{01} and GMU in PE_{10} are directly fetched from off-chip memory due to their large parameter size (with a total equal to N), while other smaller fragmented TF parameters for other GBUs and GMUs are stored in on-chip BRAMs. We reuse the same circuits for both the NTT and INTT computations.

B. GTPU

For GTPU, we propose a fine-grained dataflow based on 2D block decomposition to enable parallel block-wise small-sized transpose for optimizing critical path delay, while ensuring pipeline efficiency in subsequent NTT computations. Specifically, proposed dataflow decomposes the original N polynomial data along two dimensions n_1 and n_2 ($N = n_1 \times n_2$) into p_b=$\sqrt{p_b} \times \sqrt{p_b}$ data blocks, with each parallel block processing $p_e = \frac{p_d}{p_b}$ packed data in each cycle. To illustrate how this block-based dataflow facilitates the efficient execution of GTPU, we use a specific example with $N = 256 = 16(n_1) \times 16(n_2) = 4(n_{11}) \times 4(n_{12}) \times 4(n_{21}) \times 4(n_{22})$, $p_d = 16$, $p_b = 4$, and $p_e = 4$ to explain the GTPU with $n_1 \times n_2$ transposition in PE_{10} and $n_{11} \times n_{12}/n_{21} \times n_{22}$ transposition in PE_{00}/PE_{10}.

1) GTPU in PE_{01}: As shown in Fig.6, when p_b is set to 4, the original $N = 256$ data is decomposed along two dimensions—$n_1 = n_2 = 16$—into 2×2 blocks that are executed in parallel: $block_{00}$, $block_{01}$, $block_{10}$, and $block_{11}$. Consequently, $16(p_d)$-packed datastream is divided into four parallel $4(p_e)$-packed datastreams within each block to perform smaller-sized $p_e \times p_e$ transpositions in each TPU. The hardware structure utilizes shift-based storage and diagonal memory access, which is the same as that in Alg.2.

The output datastream of block-wise transpositions are then processed through a simple inter-block data exchange, as

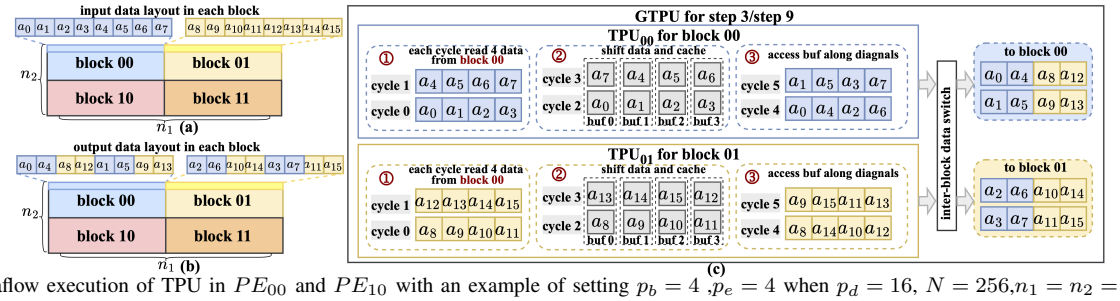


Fig. 7. Dataflow execution of TPU in PE_{00} and PE_{10} with an example of setting $p_b = 4$,$p_e = 4$ when $p_d = 16$, $N = 256$,$n_1 = n_2 = 16$, $n_{ij} = 4$.

shown in the following equation, to complete the full $n_1 \times n_2$ transposition:

$$\begin{bmatrix} block_{00} & block_{01} \\ block_{10} & block_{11} \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} block_{00}^{\mathsf{T}} & block_{10}^{\mathsf{T}} \\ block_{01}^{\mathsf{T}} & block_{11}^{\mathsf{T}} \end{bmatrix} \tag{2}$$

When larger p_b is set to achieve a smaller block for larger polynomial sizes, each block in the figure is further recursively split into 2×2 smaller sub-blocks. The complete transposition is then accomplished using a multi-level cascaded data switch structure. Each TPU employs double buffering to overlap execution latency across multiple N-sized polynomial limbs, thereby maximizing throughput.

Compared with the originally large-sized $n_1 \times n_2$ TPU in the *BaseTNTT*, our GTPU only adopts a simple inter-block data switch without introducing any additional storage cost under the same data parallelism p_d. More importantly, the subsequent NTT computations remain confined within independent blocks and are not disrupted. The specific configuration of p_b is determined by balancing the transposition granularity with its impact on the achievable deployment frequency, as discussed in Sec.III-B.

2) GTPU in PE_{00}/PE_{10}: Taking step 3 in PE_{00} as an example, when $N = 256$, PE_{00} needs to perform $n_{11} \times n_{12}$ transposition on every 16 data (i.e., a_0-a_{15} in Fig.7(a)), which corresponds to a 4×4 transposition in this case. With the proposed block-level parallel dataflow, this operation can be carried out as illustrated in Fig.7(c) without disrupting the subsequent NTT_4 ($n_{12} = 4$) execution in step 4.

Firstly, TPU_{00} takes two clock cycles to sequentially read a_0–a_3 and a_4–a_7 when $p_b = 4$ and $p_e = 4$. At the same time, TPU_{01} reads a_8–a_{11} and a_{12}–a_{15} each cycle. Next, TPU_{00} and TPU_{01} independently perform transpositions similar to the shift-store and diagonal-access method described in Alg.2. During this process, TPU_{00} output a_0, a_4, a_2, a_6 in cycle 4, followed by a_1, a_5, a_3, a_7 in cycle 5. Simultaneously, TPU_{01} obtain $a_8, a_{14}, a_{10}, a_{12}$ and $a_9, a_{15}, a_{11}, a_{13}$ in the same two cycles. Finally, the data from TPU_{00} and TPU_{01} utilizes only a simple cycle-level data switch to ensure that the output of each block maintains the correct data sequence for the subsequent NTT_4 execution, as shown in Fig.7(b).

TPU_{10} for $block_{10}$ and TPU_{11} for $block_{11}$ are identical to those for $block_{00}$ and $block_{01}$ and therefore are omitted in this description. Similarly to the GTPU in PE_{01}, GTPU for PE_{00} and PE_{10} with larger p_b configuration are achieved by recursively splitting each block into 2×2 sub-blocks. Therefore, with the fine-grained data flow in the GTPU for PE_{00} and PE_{10}, the subsequent $\text{NTT}_{n_{ij}}$ can directly obtain $4(p_2)$-packed transposed data with negligible latency overhead.

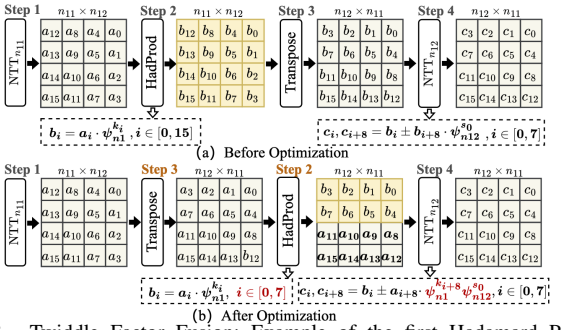


Fig. 8. Twiddle Factor Fusion: Example of the first Hadamard Product (HadProd) in TNTT, where $n_{11} = n_{12} = 4$. $\psi_{n_1}^{k_i}$ and $\psi_{n_{12}}^{s_i}$ represent the twiddle factors for Hadamard product and NTT, respectively.

C. GMU

1) Twiddle Factor Fusion Algorithm: As demonstrated in Sec.III-C, the Hadamard products in TNTT significantly consume computational resources, limiting the number of BUs deployed on FPGA. To mitigate this, we propose a cost-free optimization in which the method fuses the mod-mults of the Hadamard product and the small-point NTT, effectively reducing the total number of mod-mults in TNTT.

Fig. 8 illustrates the steps 1 to 4 of TNTT, using the first Hadamard product in TNTT as an example to demonstrate the fusion effect. Prior to optimization, the Hadamard product (step 2) performs mod-mult on each matrix element. After the transpose (step 3), the first stage of $NTT_{n_{12}}$ (step 4) is computed, where elements b_8 to b_{15} are multiplied by the twiddle factors (TFs) of NTT. Here, k_i and s_i are the exponents used to compute TFs. We observe that elements a_8 to a_{15} in the output of step 1 undergo two consecutive mod-mults: one with the Hadamard product and another with the TF $\psi_{n_{12}}^{s_0}$ of the first stage of $NTT_{n_{12}}$.

Therefore, we can pre-multiply the two sets of TFs to reduce the number of mod-mults during actual computation. The optimized flow is shown in Fig. 8(b). We first swap the order of the transpose and Hadamard product, enabling step 2 to perform mod-mults only on a_0 to a_7, while a_8 to a_{15} are bypassed directly to the output. In step 4, a_8 to a_{15} are multiplied by the fused TFs $\psi_{n_1}^{k_{i+8}}\psi_{n_{12}}^{s_0}$. This optimization halves the modular multiplications for a single Hadamard product and applies to all such computations in a multi-step NTT, resulting in a 50% reduction in the total Hadamard-related multiplication count.

Our fusion scheme maintains the same total number of TFs as the baseline design. It halves the TFs required for

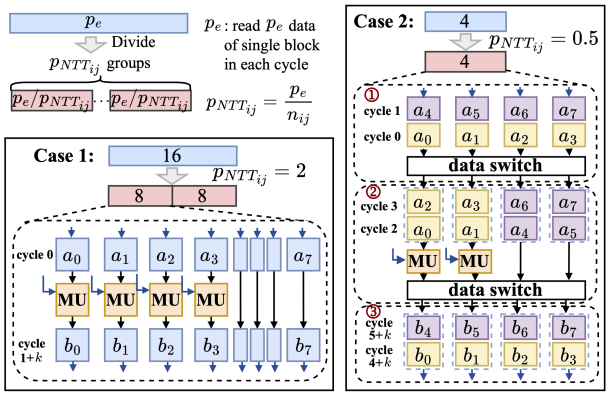


Fig. 9. Hardware architecture of GMU with an example of setting Case 1: p_e=16 and Case 2: $p_e = 4$ when $n_{ij} = 8$.

Algorithm 3: Pseudocode of GBU for each PE

```
1  for m = 0 to N/p_d do
2      #Pragma HLS PIPELINE
3      for x_0 = 0 to sqrt(p_b) do
4          #Pragma HLS UNROLL
5          for y_0 = 0 to sqrt(p_b) do
6              #Pragma HLS UNROLL
7              for n = 0 to ⌈p_NTT_nij⌉ do
8                  #Pragma HLS UNROLL
9                  for z_0 = 0 to log2 n_ij do
10                     #Pragma HLS UNROLL
11                     for z_1 = 0 to p_e / (2⌈p_NTT_nij⌉) do
12                         #Pragma HLS UNROLL
13                         BU using 2 data in p_e-datastream;
```

the Hadamard product (Fig. 8(b)) and pre-fuses the remaining half with the first-stage NTT TFs, while preserving the overall TF count unchanged. Additionally, since the proposed fusion optimization is mathematically equivalent to the original algorithm, it ensures the correctness of the algorithm while simultaneously improving computational efficiency.

2) Hardware design for GMU: Based on the proposed fusion algorithm, resource-efficient GMU is designed to perform mod-mults of Hadamard product computations in TNTT. Fig. 9 illustrates the data layout of GMU and the hardware architectures. In the figure, p_e represents the number of parallel input data within each block after 2D decomposition, consistent with the data of GTPU in Fig. 6. In the TNTT algorithm, the output of Hadamard products is grouped into n_{ij} data for subsequent $\text{NTT}_{n_{ij}}$ computation. To match the computation of $\text{NTT}_{n_{ij}}$, the p_e data are divided into $p_{\text{NTT}_{n_{ij}}} = p_e/n_{ij}$ groups. Each group contains an equal number of mod-mult-based MUs. Our design allows flexible parallelism configuration to adapt to various N in FHE. The relationship between the block parallelism p_e and the TNTT parameter n_{ij} presents two different scenarios that directly shape the hardware design.

Case 1 shows the scenario where $p_e \geq n_{ij}$. For instance, with $p_e = 16$, $n_{ij} = 8$, and $p_{\text{NTT}_{n_{ij}}} = 2$, the 16 inputs are divided into two groups, each containing $n_{ij} = 8$ data. The optimizations in Sec.IV-C1 demonstrate that, during the Hadamard product computation, only the first half of the n_{ij} numbers require mod-mults, while the second half bypasses computation. This reduction halves the required MUs directly in the GMU design: only the first four inputs a_0-a_3 in Fig. 9 are assigned MUs, reducing the total from 8 to 4, and optimizing hardware cost.

Case 2 considers scenarios where $p_e < n_{ij}$. In this case, multiple cycles are required to transfer n_{ij} data. For example, with $p_e = 4$, $n_{ij} = 8$, and $p_{\text{NTT}_{n_{ij}}} = 0.5$, it takes 2 cycles to obtain 8 data while reading 4 data each cycle. According to Fig. 9, the data input during cycle 0 (a_0-a_3) performs mod-mults, while that in cycle 1 (a_4-a_7) does not. As a result, the optimization of halving the MUs cannot be directly applied. Therefore, we introduce extra data switch units. After the n_{ij} input data are obtained, the data switch unit is employed to reconfigure the data layout. Specifically, mod-mults of a_0-a_3 are distributed across two cycles. After computation, another data switch restores the data to its original layout. Since the n_{ij} in TNTT are typically small (e.g., 8, 16, etc.) and p_e is smaller, the overhead introduced by the data switch is minimal.

The optimized architecture incurs no extra storage or bandwidth overhead in TFs. This is because the number of TFs remains unchanged, and the bandwidth required for reading TFs—specifically, p_d per cycle—is preserved, with $p_d/2$ TFs allocated to the Hadamard product and $p_d/2$ fusioned TFs to the subsequent NTT stage. Furthermore, our design remains effective for pipelining NTT limbs with varying modulus chains in FHE.

D. GBU

Alg.3 presents the fine-grained dataflow of executing $NTT_{n_{ij}}$ for the GBU in each PE_{ij}. Following proposed 2D block decomposition dataflow for GTPU, GBU also splits the total data parallelism p_d along the n_1 and n_2 dimensions into unrolled $\sqrt{p_b} \times \sqrt{p_b}$ blocks, as highlighted by GREEN in lines 3–6. Each block unroll $\lceil p_{NTT_{n_{ij}}} \rceil$ parallel $NTT_{n_{ij}}$ instances with p_e-packed datastream, as highlighted by ORANGE in lines 7–13. Similar to the discussion in GMU, the configuration of p_e has two different scenarios in GBU depending on the n_{ij} for different N. When **Case 1**: $p_e < n_{ij}$ and $\lceil p_{NTT_{n_{ij}}} \rceil$ = 1, each block instantiates one $NTT_{n_{ij}}$. All $\log_2 n_{ij}$ stages of $NTT_{n_{ij}}$ are fully unrolled in line 9, with each stage unrolling $\frac{p_e}{2}$ BUs in line 11. When **Case 2**: $p_e < n_{ij}$, the unroll number of $NTT_{n_{ij}}$ in each block is $p_{NTT_{n_{ij}}}$ in line 9, and all $\log_2 n_{ij}$ stages are unrolled and each stage unrolls all BUs, which corresponds to $\frac{p_e}{2\lceil p_{NTT_{n_{ij}}} \rceil} = \frac{n_{ij}}{2}$ in line 11. Each BU in GPU supports optimized Barrett reduction [5], Montgomery reduction [27], and word-level Montgomery reduction(WLM) for NTT-friendly primes [26].

E. Architectural Scalability on Multi-Step NTTs

The proposed architecture is generalizable to other multi-step NTTs, such as the 7-step and 13-step variants, as they all share the fundamental challenge of managing intermediate transposition and Hadamard product operations in multi-step NTT decompositions. As illustrated in Fig.5, the hardware

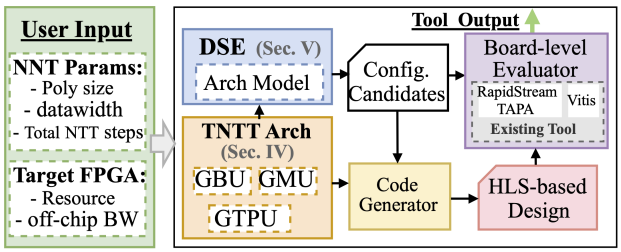


Fig. 10. Design automation tool of proposed AutoNest.

TABLE II. Configurable parameters for proposed architecture.

Variable	Explanation
p_d	Total data parallelism
p_b	The number of parallel blocks
p_e	Data parallelism in each block
$p_{NTT_{n_{ij}}}$	The number of $NTT_{n_{ij}}$ in each block

implementation for a given multi-step NTT maps each step—small-point NTT, transposition, and Hadamard product—to independent GBUs, GTPUs, and GMUs, respectively. This mapping can yield a fully pipelined hardware structure.

V. Design Space Exploration for AutoNest

Building upon the proposed highly scalable dataflow architecture, we further developed a complete automation tool–AutoNest–to enable systematic design space exploration for identifying the optimal TNTT implementation that maximizes throughput for given NTT parameters and FPGA device.

As illustrated in Fig.10, AutoNest takes the NTT parameters, including polynomial size and data width, as well as target device, which includes available DSPs, LUTs, BRAMs, FFs, and off-chip bandwidth as input. The flexibility of the proposed hardware architecture allows the total number of NTT steps to be a configurable input parameter, enabling support for TNTT and other multi-step variants. Based on the architectural analytical model discussed in Section V-A, it first generates a set of design candidates with different architectural parameter configurations by maximizing data parallelism within the resource constraints. A code generator then produces HLS-based design code for these candidates. Finally, a performance evaluator on the target FPGA board is utilized to identify the optimal design that achieves the highest throughput (and thus the lowest average latency) among the candidates. Our evaluator integrates RapidStream-TAPA tool [1], which is built upon Xilinx Vitis [32], to express task-parallel designs and efficiently explore optimal floorplanning solutions for improved resource utilization and performance.

A. Architecture Modeling

Configurable parameters for our architecture are summarized in Table II. The accelerator supports typical FHE parameters, with N values ranging from 2^{12} to 2^{16} and bit-widths $\log q$ from 28-bit to 64-bit. Based on these variables, our analytical model for TNTT and other multi-step NTT variants is as follows.

Average Latency (AL): To reflect the achieved throughput, the average latency for processing a single polynomial is defined and computed as follows:

$$AL = \frac{N}{p_d \cdot f} \tag{3}$$

where f denotes the clock frequency achievable by our architecture when configurable with parameters p_d, p_b and p_e.

DSP Resources (D): The DSP consumption of TNTT or other multi-step NTT is determined by the number of mod-mults in the GMU and GBU of all PEs, which is calculated as follows:

$$\begin{aligned} D &= \sum_{PE} D_1(GMU) + D_2(GBU) \\ &= (\frac{S_h \cdot p_d}{2} + \frac{p_d}{2} log_2 N) \cdot d_{const} \\ &= p_d \cdot D_{const} \end{aligned} \tag{4}$$

where the second row of left-hand term represents the total number of mod-mults in the GMUs. S_h denotes the number of GMUs for the Hadamard product steps in the multi-step NTT. For instance, this term for TNTT is $\frac{3p_d}{2}$ because TNTT has three GMUs ($S_h = 3$) in PE_{00}, PE_{01}, and PE_{10}, with each GMU contains $\frac{p_d}{2}$ mod-mults after proposed fusion optimization in Sec.IV-C1. The right-hand term corresponds to the total number of mod-mults in the GBUs of all PEs. The constant d_{const} represents the number of DSPs consumed by single mod-mult when utilizing different modular reduction methods for a given bit-width $\log q$.

BRAM Resources (B): BRAM consumption mainly originates from the TPUs. For instance, since GTPUs in PE_{00} and PE_{01} processes smaller-scale transposition with $n_{11} \times n_{12}$ and $n_{21} \cdot n_{22}$, TNTT mainly consumes LUTs or a small amount of BRAM, and its contribution is omitted in our model. BRAM consumption is primarily attributed to large-scale transpositions, such as the $n_1 \times n_2$ transposition in $GTPU_{01}$ of TNTT. Therefore, BRAM consumption for multi-step NTT can be computed as follows:

$$B = \sum_{PE} B(GTPU) \approx 4 \cdot p_d \cdot \left\lceil \frac{\log q}{width} \right\rceil \cdot \left\lceil \frac{N}{p_d \cdot depth} \right\rceil \tag{5}$$

where $width$ and $depth$ refer to the bitwidth and depth of a single BRAM in the target FPGA, respectively.

LUT and FF Resource (LUT and FF): By examining the architecture in Sec.IV, it can be observed that the basic components of the architecture include TPUs with different sizes, a mod-mul operator, and some low-overhead data-switch logic. Therefore, we collected a set of LUT and FF resource usage for these basic components under various sizes and data widths using actual Vivado synthesis reports, and constructed a linear model based on these pre-collected data. Once the parameter configuration of the complete architecture is determined, the overall LUT and FF resource consumption can be obtained by summing the resource usage of these basic components as predicted by the linear model.

Average Off-chip Memory Bandwidth (ABW): We configure separate off-chip memory access interfaces for the polynomial input data stream, output data stream, and the TF parameters. Therefore, the average total bandwidth for a single polynomial limb can be estimated as follows:

$$ABW = \frac{3 \cdot N \cdot \log q}{AL} = 3 \cdot p_d \cdot \log q \cdot f \tag{6}$$

Algorithm 4: Design space exploration

Input: NTT params:N with total NTT steps: S
FPGA info: D_{max},B_{max}, LUT_{max},FF_{max}, ABW_{max};
Output: Optimal solution with AL_{min};

1 $p_{d_{min}} \leftarrow 8$(Eq.(7)); $p_{d_{max}} \leftarrow \frac{D_{max}}{D_{const}}$(Eq.(4));
2 **while** $p_{d_{min}} \leq p_{d_{max}}$ **do**
3 $p_{d_{cur}} \leftarrow p_{d_{min}} \times 2^{\lfloor \frac{\log_2(p_{d_{max}}/p_{d_{min}})}{2} \rfloor}$;
4 compute current $_{B_{used},LUT_{used},FF_{used}}$;
5 **if** $\{B_{used}, LUT_{used}, FF_{used}\} \leq \{B_{max}, LUT_{max}, FF_{max}\}$ **then**
6 $p_{d_{opt}} \leftarrow p_{d_{cur}}$; $p_{d_{min}} \leftarrow p_{d_{cur}} \times 2$;
7 **else**
8 $p_{d_{max}} \leftarrow p_{d_{cur}}/2$;
9 $p_{b_{min}} \leftarrow 4$; $p_{b_{max}} \leftarrow \frac{p_{d_{opt}}}{2}$;
10 **for** $p_{b_{pos}} = p_{b_{min}}$ **to** $p_{b_{max}}$ **do**
11 $p_{e_{pos}} = \frac{p_{d_{opt}}}{p_{b_{cur}}}$;
12 // Board-level evaluator in line 14-19
13 Find possible candidate with $\{p_{d_{opt}}, p_{b_{pos}}, p_{e_{pos}}\}$;
14 $f_{p_{b_{pos}}} \leftarrow$ P/R solutions searched by RapidStream-TAPA;
15 $AL_{pos} \leftarrow \frac{N}{p_{d_{opt}} \cdot f_{p_{b_{pos}}}}$ (Eq.(3)) ;
16 $ABW_{used} \leftarrow$ (Eq.(6)) ;
17 **if** $AL_{pos} \leq AL_{min}$, $ABW_{used} \leq ABW_{max}$ **then**
18 $AL_{min} \leftarrow AL_{pos}$;
19 $\{p_{b_{opt}}, p_{e_{opt}}, f_{opt}\} \leftarrow \{p_{b_{pos}}, p_{e_{pos}}, f_{p_{b_{pos}}}\}$
20 **Return** $\{AL_{min}, p_{d_{opt}}, p_{b_{opt}}, p_{e_{opt}}, f_{opt}\}$;

B. Design Space Exploration

Alg.4 presents the pseudocode for DSE. According to our model described by Eq.(3)-(6) in Sec.V-A, most performance metrics, such as resource usage, are entirely determined by the total data parallelism p_d. The average latency (AL) and bandwidth ABW, however, also depend on the achievable clock frequency of the final architecture. In this way, determining the final architecture requires solving for p_b and p_e.

To explore the monotonic relationship between AL and p_d in Eq.(3), our DSE employs an efficient binary search strategy that first identifies the optimal data parallelism $p_{d_{opt}}$ satisfying the resource constraints, including available DSPs (D_{max}), BRAMs (B_{max}), LUTs (LUT_{max}), FFs (FF_{max}) (lines 1–8). Specifically, since the core basic component of NTT is the mod-mult operator, its implementation primarily relies on cascaded DSPs under various modular reduction methods. Thus, the maximum available DSPs D_{max} are used to derive an initial rough estimate of the maximum data parallelism $p_{d_{max}}$ according to Eq.(4) in line 1. Additionally, the proposed architecture itself has the following constraints:

$$\frac{p_e}{2} \geq 1, \quad p_b \geq 4 \tag{7}$$

The first constraint requires that at least one BU is instantiated in each NTT stage of GBU, which corresponds to the product of loop iterations in lines 7 and 11 in Alg.3. The second constraint ensures that, for FHE applications where $N \geq 2^{12}$, the single polynomial limb must be at least decomposed into $p_b = 2 \times 2$ parallel blocks for better GTPU performance. Given the relationship $p_d = p_b \times p_e$, this implies that the initial minimum data parallelism is $p_{d_{min}} = 8$.

After determining the optimal $p_{d_{opt}}$, lines 9–18 further utilize the board-level evaluator to determine the optimal configuration AL_{min} with $\{p_{b_{opt}}, p_{e_{opt}}, f_{opt}\}$ while ensuring $ABW_{used} \leq ABW_{max}$. The upper and lower bounds of $p_{b_{pos}}$ including $p_{b_{min}}$ and $p_{b_{max}}$ in line 10 are derived using Eq.(7). Lines 14–19 evaluate each possible configuration candidate $\{p_{d_{pos}}, p_{b_{pos}}, p_{e_{pos}}\}$ by employing RapidStream-TAPA [1] to explore the optimal placement and routing (P/R) solution, and the achievable clock frequency is determined through actual FPGA deployment feedback.

VI. Results

A. Experimental Setup

To evaluate the effectiveness of the proposed AutoNest , we perform a performance comparison with existing SOTA works under various parameter settings, followed by a detailed analysis of GTPU, GMU, and DSE. This subsection provides a detailed description of the experimental setup.

Platforms and toolchain: The proposed architecture is evaluated on an AMD-Xilinx Alveo U280 FPGA board, which provides 9,024 DSPs, 1,304 kLUTs, 4,032 BRAMs, 960 URAMs, and 32 HBM banks (8GB in total). Vitis HLS 2023.2 is used to synthesize the DSE-generated HLS design into RTL IP, followed by placement and routing using Vivado 2023.2. The generated bitstreams are executed and evaluated on the U280 platform using Xilinx Runtime (XRT) 2023.2.

NTT parameters selection: Our experiments select several representative parameter settings, including $N = 2^{12}, 2^{14}, 2^{16}$ with $\log q = 32$ and 64. All experiments are performed on TNTT (S=10), with the sole exception of Sec. VI-E.

Performance metrics: We use average latency as the primary performance metric, which is calculated as the mean execution time of multiple consecutive N-polynomial limbs. The kernel execution time includes data transfers between off-chip memory (e.g., HBM) and on-chip memory but excludes the transfer time between the CPU and FPGA. Additionally, we introduce the Area Time Product (ATP) metric (lower is better) [24] for a fair comparison with existing different architectures.

FHE benchmarks: We extend our NTT accelerator to support FHE operations, including NTT-based homomorphic multiplication (HMult), Rescale, and Rotate. Furthermore, we assess the bootstrapping [6], [19], whose performance metric [19] is calculated as $T_{\text{Mult},a/\text{slot}} := \frac{T_{\text{Boot}} + \sum_{i=1}^{l} T_{\text{Mult}(i)}}{l \cdot \text{slot}}$.

TABLE III. Optimal configuration identified by DSE.

Param. N-$(n_{11}, n_{12}, n_{21}, n_{22})$-$\log q$	p_d	p_b	p_e	$p_{NTT_{n_{11},n_{12},n_{21},n_{22}}}$
2^{12}-(2^3,2^3,2^3,2^3)-32	128	16	8	1,1,1,1
2^{12}-(2^3,2^3,2^3,2^3)-64	32	4	8	1/1/1/1
2^{14}-(2^4,2^4,2^3,2^3)-32	64	4	16	1/1/2/2
2^{14}-(2^4,2^4,2^3,2^3)-64	32	4	8	0.5/0.5/1/1
2^{16}-(2^4,2^4,2^4,2^4)-32	64	4	16	1/1/1/1
2^{16}-(2^4,2^4,2^4,2^4)-64	32	4	8	0.5/0.5/0.5/0.5

B. Comparison with State-of-the-art

Table III gives the optimal parameter configuration for typical FHE parameters. The experimental results in Table IV demonstrate that AutoNest outperforms state-of-the-art

TABLE IV. Comparison with previous work.

Method	N-log q	Device	kLUT/kFF/ BRAM/URAM/DSP	#BU	Freq (MHz)	Avg Lat. (us)	Throughtput (NTT/s)	ATP ↓
Proteus [18][2,4]	2^{12}-32	Virtex7	8/4/8/0/44	–	150	13.8 (86.3x)	72,464	177
AutoNTT [24][1,4]	2^{12}-32	U280	338/377/34/96/2,034	384	215	0.34 (2.13x)	2,941,176	260
IO [22][2,4]	2^{12}-32	U280	137/160/40/0/1,631	384	250	0.52 (3.25x)	1,923,077	205
Ours[1,2,4]	2^{12}-32	U280	652/814/228/0/5,760	768	245	**0.16** (1.00x)	**6,250,000**	239
Proteus [18][2,4]	2^{12}-64	Virtex7	23/18/16/0/220	–	150	13.8 (27.60x)	72,464	765
ESC-NTT [17][2,4]	2^{12}-64	U280	523/1,478/275/0/6,518	–	300	3.81 (7.62x)	262,467	6,701
AutoNTT [24][1,4]	2^{12}-64	U280	393/655/68/64/3,840	192	250	0.62 (1.24x)	1,612,903	678
IO [22][2,4]	2^{12}-64	U280	356/376/72/0/5,040	192	250	0.52 (1.04x)	1,923,077	608
Ours[1,2,4]	2^{12}-64	U280	527/909/124/0/4,640	192	292	**0.5** (1.00x)	**2,000,000**	619
AutoNTT [24][1,4]	2^{14}-32	U280	415/465/68/160/2,688	224	250	1.76 (1.39x)	568,182	1,730
IO [22][2,4]	2^{14}-32	U280	94/112/48/0/1,056	112	250	4.14 (3.28x)	241,546	1,085
Ours[1,2,4]	2^{14}-32	U280	430/579/128/3,264	448	280	**1.26** (1.00x)	**793,651**	1,112
FAB [3][3]	2^{14}-54	U280	–	256	300	5.99 (2.61x)	166,945	–
AutoNTT [24][1,4]	2^{14}-64	U280	453/728/68/96/4,480	224	248	2.49 (1.09x)	401,606	3,235
IO [22][2,4]	2^{14}-64	U280	234/256/96/0/3,280	112	250	4.14 (1.81x)	241,546	3,201
Ours[1,2,4]	2^{14}-64	U280	650/844/188/0/5,440	224	298	**2.29** (1.00x)	**436,681**	3,262
Poseidon [35][2,3]	2^{16}-32	U280	358/344/1024/0/4,032	512	450	80.17 (22.3x)	12,469	92,320
AutoNTT [24][1,4]	2^{16}-32	U280	479/460/100/224/3,072	256	161	8.8 (2.45x)	113,636	10,390
IO [22][2,4]	2^{16}-32	U280	170/192/152/0/2,016	256	250	8.28 (2.31x)	120,773	4,188
Ours[1,2,4]	2^{16}-32	U280	476/638/252/0/3,648	512	285	**3.59** (1.00x)	**278,552**	3,600
OpenNTT [23][2]	2^{16}-52	ZCU102	157/117/98/0/896	–	210	78.4 (9.24x)	12,755	19,882
SAM [31][2]	2^{16}-64	U250	267/328/2126/0/2,736	–	165	380 (44.81x)	2,632	414,770
AutoNTT [24][1,4]	2^{16}-64	U280	503/840/292/128/5,120	256	208	10.01 (1.18x)	99,900	15,574
IO [22][2,4]	2^{16}-64	U280	460/470/280/0/6,320	256	248	**8.34** (0.98x)	**119,904**	12,560
Ours[1,2,4]	2^{16}-64	U280	747/929/176/184/6,080	256	280	8.48 (1.00x)	117,925	15,950

[1]: HLS design; designs without the mark are RTL. [2]: Supports multiple N and q values; designs without the mark are fixed. [3]: Iterative NTT architecture optimized for latency; others are dataflow, optimized for throughput. [4]: Uses WLM mod-mult; FAB uses WK, Poseidon uses Barrett. $ATP = AL \times (kLUT + kFF + DSP + BRAM + 8 \times URAM)/5$. The results of the existing works are as reported in their respective papers, except for AutoNTT (reproduced).

solutions, achieving an average speedup of 2.31× on the same U280 FPGA, with most designs achieving clock frequencies of 280∼298 MHz. Specifically, among recent NTT architectures, AutoNTT [24] (based on the SNTT algorithm and implemented with dataflow designs in HLS) and IO [22] (using the TNTT with dataflow architecture in RTL) serve as the primary benchmarks to support different FHE parameters. Under the same mod-mult design, our design achieves a 1.43× ATP improvement and a 1.58× speedup on average. Compared to IO, our design also achieves a 2.1× average speedup. For different degrees of polynomials $N = 2^{12}, 2^{14}, 2^{16}$ with $\log q = 32$ bitwidth, we deploy 2×∼4× more BUs (768, 448, and 512, respectively), resulting in an average speedup of 2.47× over these two works. For 64-bit q, resource usage increases significantly due to larger bit-width mod-mult, limiting the number of BUs to match AutoNTT [24] and IO [22]. Proposed parallel block-wise transpose strategy optimizes the critical path, achieving better frequencies of 280∼298 MHz along with 1.22× average speedup.

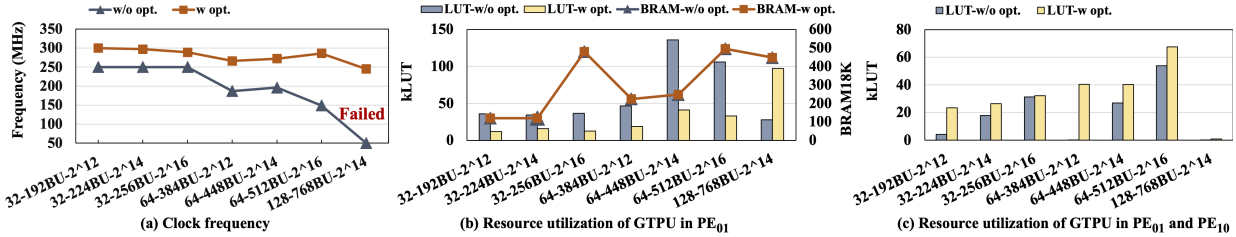
Proteus [18] and OpenNTT [23] generate smaller designs with only a small number of BUs. In comparison, our accelerator achieves higher frequency and better average latency while maintaining a similar order of ATP. Other accelerators limit supported NTT parameters. For example, ESC-NTT [17] only supports $N = 2^{12}$, while SAM [31] does not support parameter configurations smaller than 2^{16}. Furthermore, our accelerator presents significant improvements in both average latency and ATP compared to ESC-NTT [17] and SAM [31].

We also compare with recent Poseidon [35] and FAB [3] FHE accelerators, with fixed-parameter NTT as their core optimization unit. For $N = 2^{14}$, our dataflow architecture with $\log q = 64$ bit achieves 2.61× speedup over FAB with $\log q = 54$. Compared to Poseidon for $N = 2^{16}$ and $\log q = 32$, our architecture delivers a 22.3× speedup.

C. Performance Breakdown

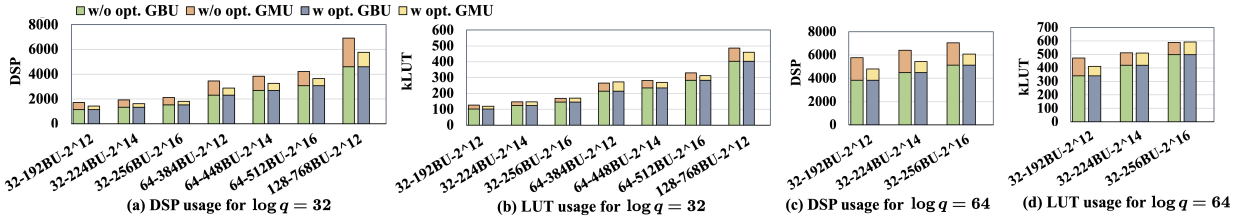
1) Transpose Optimization Analysis: Fig.11(a) illustrates the clock frequency of AutoNest before and after transpose optimization across different data parallelism p_d configurations for different N. As observed in Fig.11(a), by employing proposed 2D decomposition-based strategy for block-wise transpose optimization, we achieve a frequency improvement of 39∼137 MHz under different parallelism settings. For $N = 2^{12}$ with $p_d = 128$, the proposed optimization effectively mitigates the routing failure caused by terrible critical path delay, resulting in a clock frequency of 245 MHz.

GTPU in PE_{01}: Fig.11(b) presents the comparison of LUT and BRAM resource utilization for the $n_1 \times n_2$ transpose in PE_{01} before and after optimization. Specifically, the optimized design demonstrates an average 2.86× LUT reduction



(a) Clock frequency (b) Resource utilization of GTPU in PE_{01} (c) Resource utilization of GTPU in PE_{01} and PE_{10}

Fig. 11. Performance comparison before and after GTPU optimization. p_d-B-C refers to data parallelism p_d with B BUs when $N = C$ and $\log q = 32$.



(a) DSP usage for $\log q = 32$ (b) LUT usage for $\log q = 32$ (c) DSP usage for $\log q = 64$ (d) LUT usage for $\log q = 64$

Fig. 12. Resource comparison before and after GMU optimization.p_d-B-C refers to data parallelism is set to p_d with B BUs for $N = C$.

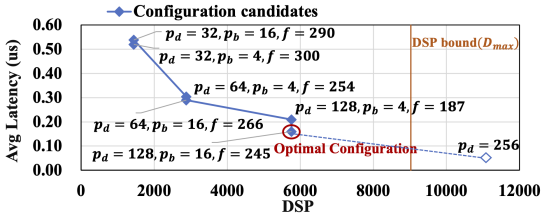
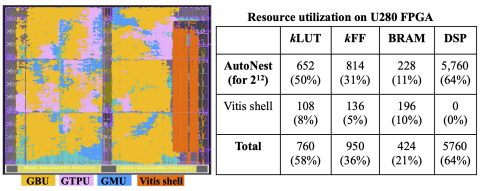


Fig. 13. Design space exploration for $N = 2^{12}, \log q = 32$ on U280 FPGA.



Resource utilization on U280 FPGA

	kLUT	kFF	BRAM	DSP
AutoNest (for 2^{12})	652 (50%)	814 (31%)	228 (11%)	5,760 (64%)
Vitis shell	108 (8%)	136 (5%)	196 (10%)	0 (0%)
Total	760 (58%)	950 (36%)	424 (21%)	5760 (64%)

Fig. 14. Implementation layout and resource utilization of AutoNest with $p_d = 128$, $p_b = 16$ for $N = 2^{12}$ on AMD-Xilinx U280 FPGA.

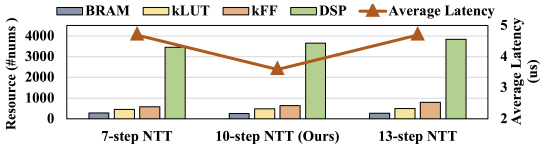


Fig. 15. AutoNest performance on multi-step NTTs.

TABLE V. DSE-identified optimal configuration for multi-step NTTs.

N-$\log q$	Total Steps	NTT size	p_d	p_b	p_e
2^{16}-32	7	$2^6/2^5/2^5$	64	4	16
	13	$2^4/2^3/2^3/2^3/2^3$			

for $p_d = 32$ and $p_d = 64$ compared to the unoptimized counterpart. This reduction is primarily attributed to our block-wise transpose strategy, where each smaller segmented transpose (i.e., $p_b = 4$ for $p_b = 32$ and $p_b = 64$ for $N = 2^{12}$,$N = 2^{14}$,$N = 2^{16}$ in Table III) incurs significantly lower addressing logic overhead within each TPU. For the case of $p_d = 128$ with $N = 2^{12}$, while individual smaller transpose size (i.e., $p_e = 8$) exhibit low resource overhead, the optimal block number $p_b = 16$ leads to an overall increase in LUT resources. However, this increase is justified as it effectively optimizes the critical path delay and resolves the routing failures in the unoptimized design, achieving a clock frequency of 245 MHz. Additionally, proposed block-based transpose decomposition does not introduce any additional memory overhead as described in Sec.IV-B1. Therefore, the BRAM consumption remains consistent between the optimized and unoptimized designs.

GTPU in PE_{01} and PE_{01}: Fig.11(c) summarizes the performance comparison of the $n_{11} \times n_{12}$ transpose in PE_{00} and the $n_{21} \times n_{22}$ transpose in PE_{10} before and after optimization. The transposes in PE_{00} and PE_{10} involve relatively smaller matrix sizes, including 8×8 and 16×16 for $N = 2^{12}$ to 2^{16}. LUT consumption in these two PEs includes not only the logic for data reorganization but also the storage of data. Although the implementation of the parallel block-based transpose results in increased LUT utilization compared to the unoptimized design, this overhead is a necessary trade-off. It ensures that the data output from the GTPU is seamlessly streamed into the subsequent GMU and GBU, thereby facilitating the efficient pipeline execution of mod-mult operations.

2) Hadamard Products Optimization Analysis: Fig.12 presents the resource proportion of the GMUs for executing Hadamard product before and after twiddle factor fusion optimization. As shown in Fig.12(a) and Fig.12(c), For $N = 2^{12}$, 2^{14}, and 2^{16} with different $\log q$-bit widths, the DSP usage percentage of GMU decreases by 16.7%, 15%, and 13.6%, respectively, which is consistent with the analysis in Eq.(4). In terms of absolute DSP usage, for $p_d = 128$, the optimization reduces the DSPs by 1,152, which accounts for 14% of the available DSP resources on the U280 FPGA. This demonstrates that the proposed fused optimization provides increasingly significant resource-saving benefits as the parallelism increases. As shown in Fig.12(b) and Fig.12(d), LUT resource utilization slightly decreases in most cases, primarily due to the reduced number of mod-mul operators within the GMU. For certain cases, such as $p_d = 32$ with $N = 2^{14}$ or 2^{16}, a slight increase in LUT usage is observed. This is mainly caused by the additional data switch logic (Fig.9: Case 2) introduced when $p_{NTT_{n_{ij}}} < 1$.

D. Design Space Exploration

Our DSE is capable of determining feasible design parameter configurations, including p_d, p_b and f, under given FPGA resource constraints. It then generates the corresponding

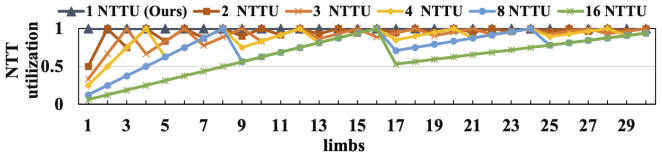


Fig. 16. Utilization of NTTU copies with same resources across different limbs.

HLS-based design for these parameter configurations and performs on-board evaluations to feed back to us the optimal architectural parameters. Fig.13 illustrates an example of a design space for $N = 2^{12}$ and $\log q = 32$. As shown in Fig.13, the same p_d achieves a similar measured average latency. DSE finds $p_d = 128$ due to avaliable resource on the U280 FPGA. In the same p_d, it provides multiple candidates for p_b. After conducting an actual hardware evaluation to balance the impact of the number of transpose tiles (p_b) and tile size (p_e) on operating frequency f, $p_b = 16$ is selected, achieving an average latency of 0.16 us with 245 MHz frequency. Fig.14 further presents the implementation layout of the optimal design for this example.

E. Scalability on Multi-step NTT

To validate the scalability of AutoNest across different-step decomposition, Fig.15 presents performance evaluation for 7-step and 13-step NTT with parameters $N = 2^{16}$ and $\log q = 32$ on a U280 FPGA. The optimal configurations found by DSE are listed in Table V. Both 7-step and 13-step NTT achieve the same maximum parallelism p_d = 64 with 512 BU as TNTT, as shown in Table III. The overall DSP usage exhibits an upward trend from the 7-step to the 13-step NTT. This is attributed to the increased Hadamard product steps and consequently more instantiated GMUs as the decomposition step grows. Furthermore, with the proposed 2D block decomposition for large-scale transpositions, both the 7-step and 13-step designs achieved clock frequencies of 220 MHz. Notably, TNTT for $N = 2^{16}$ and $\log q$=32 yields higher average latency than its 7-step and 13-step counterparts, reflecting the inherent trade-off between the number of decomposition steps and achievable hardware performance gains.

F. NTT Efficiency Comparison with NTT-level Parallelism

In FHE applications, the NTT limb count varies with multiplication depth (typically 1-30), requiring parallelization strategies that remain efficient across this dynamic range. Fig. 16 compares the NTT utilization of our intra-NTT parallelism method—where parallelism is configured within a single NTT instance—and the existing inter-NTT strategy [25], which uses multiple parallel NTT copies, across varying limb settings. The inter-NTT approach frequently results in idle compute units when the limb count is not a multiple of the copy number. For example, when processing 5 limbs with 4 NTT copies, 3 copies are idle in the second round, reducing efficiency. In contrast, the intra-NTT pipeline maintains full utilization regardless of limb count, making it widely adopted in existing FHE accelerators [3], [35].

TABLE VI. Resource comparison on U280 FPGA for parameter Set-A: $N = 2^{16}$, $\log q = 54$, $L = 24$; Set-B: $N = 2^{16}$, $\log q = 32$, $L = 44$.

Param.	Scheme	kLUT	kFF	BRAM	URAM	DSP
Set-A	FAB [3]	899	2073	3840	960	5120
	Ours	870	1198	1920	576	6464
Set-B	Poseidon [35]	728	915	2048	/	8640
	Ours	693	792	1276	768	7360

TABLE VII. FHE operation latency (ms).

Param.	Set-A		Set-B	
Scheme	FAB [3]	**Ours**	Poseidon [35]	**Ours**
HMult	1.71 (1.15×)	1.49	3.66 (2.44×)	1.50
Rescale	0.19 (1.12×)	0.17	0.25 (1.56×)	0.16
Rotate	1.57 (1.38×)	1.14	3.31 (2.80×)	1.18

TABLE VIII. Bootstrapping time (us) computed in $T_{\text{Mult},a/\text{slot}}$.

Scheme	FAB [3]	Poseidon [35]	**Ours**
Bootstrapping	0.477 (1.22×)	0.840 (2.16×)	0.389

G. Performance Evaluation of FHE with AutoNest

Based on AutoNest, we extend other fundamental FHE operators—Base Conversion, Inner Product, and Automorphism—on a U280 FPGA to enable end-to-end FHE performance evaluation. These operators utilize a fully-pipelined architecture with the same data parallelism p_d as the NTT. The memory organization and dataflow are inspired by MAD [2] and FAB [3]. The resource usage of accelerator is summarized in Table VI. For FHE operations in Table VII, the proposed design achieves speedups of 1.12–1.38× over FAB [3] and 1.56–2.80× over Poseidon [35]. The Rotate operation exhibits the most significant speedup, which is attributable to its high proportion of NTT computations and the direct benefits from our NTT performance enhancements. Furthermore, at the application level, we evaluate the bootstrapping in TableVIII and achieve speedups of 1.22× and 2.16× against FAB and Poseidon, respectively.

VII. Realted Work

In recent years, extensive FPGA-based NTT accelerators have been proposed [3], [18], [22]–[24]. Among them, FAB [3] and Poseidon [35] implemented traditional SNTT-based acceleration for fixed parameters N and $\log q$. In contrast, some works [18], [22], [24], [31], [34] explored flexible hardware architectures to support varying N and $\log q$. Notably, SAM [31] and [22] implement fully-pipelined hardware structures by directly mapping multi-step NTT algorithms. Beyond FPGA-based acceleration, several studies have investigated GPU [14], [36] and ASIC-based accelerators [20], [28] that also leveraged multi-step NTT to enable higher parallelism. For instance, the GPU-based WarpDrive [14] applied ten-step NTT decomposition to significantly reduce instruction count and pipeline stalls. The ASIC-based F1 [28] introduced a four-step NTT pipeline to effectively save on-chip bandwidth. Unlike prior works, AutoNest enhances multi-level NTT with algorithm/hardware co-optimization, and incorporates automated exploration of parameter configurations.

VIII. Conclusion

This paper introduces AutoNest, a scalable and efficient FPGA-based TNTT accelerator for FHE. Leveraging algorithm-hardware co-optimizations—including a 2D block decomposition dataflow and a cost-free twiddle factor fusion, our fully pipelined architecture improves the clock frequency and data parallelism of the TNTT accelerator. Furthermore, AutoNest features an automatic DSE framework to efficiently generate HLS-based accelerators for diverse NTT parameters. Experiments on an AMD-Xilinx U280 FPGA show that AutoNest achieves a 2.31× average speedup over SOTA designs, thereby enabling more efficient acceleration of FHE systems.

References

[1] "Rapidstream tapa compiles task-parallel hls program into high-frequency fpga accelerators." https://github.com/rapidstream-org/rapidstream-tapa, 2024.

[2] R. Agrawal, L. de Castro, C. Juvekar, A. P. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "MAD: memory-aware design techniques for accelerating fully homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 2023, pp. 685–697.

[3] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[4] D. H. Bailey, "Ffts in external or hierarchical memory," *The journal of Supercomputing*, vol. 4, no. 1, pp. 23–35, 1990.

[5] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[6] J. Bossuat, C. Mouchet, J. R. Troncoso-Pastoriza, and J. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, 2021, pp. 587–617.

[7] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual cryptology conference*. Springer, 2012, pp. 868–886.

[8] 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.

[9] J. H. Cheon, K. Han, A. Kim, and et al., "A full RNS variant of approximate homomorphic encryption," in *SAC*, 2018.

[10] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International conference on the theory and application of cryptology and information security*. Springer, 2017, pp. 409–437.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT*, vol. 10031, 2016, pp. 3–33.

[12] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[13] X. Deng, S. Fan, Z. Hu, Z. Tian, Z. Yang, J. Yu, D. Cao, D. Meng, R. Hou, M. Li *et al.*, "Trinity: A general purpose fhe accelerator," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 338–351.

[14] G. Fan, M. Zhang, F. Zheng, S. Fan, T. Zhou, X. Deng, W. Tang, L. Kong, Y. Song, and S. Yan, "Warpdrive: Gpu-based fully homomorphic encryption acceleration leveraging tensor and cuda cores," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 1187–1200.

[15] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, p. 144, 2012.

[16] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 922–934.

[17] Z. Guan, Y. Zhu, Y. Huang, L. Lei, X. Wang, H. Jia, Y. Chen, B. Zhang, J. Dong, and S. Bian, "Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[18] F. Hirner, A. C. Mert, and S. S. Roy, "Proteus: A pipelined ntt architecture generator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 7, pp. 1228–1238, 2024.

[19] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 4, pp. 114–148, 2021.

[20] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[21] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1237–1254.

[22] E. Koçer, S. Kirbiyik, T. Tosun, E. Alaybeyoglu, and E. Savas, "Io-optimized design-time configurable negacyclic seven-step ntt architecture for fhe applications," in *Proceedings of the Great Lakes Symposium on VLSI 2025*, 2025, pp. 14–21.

[23] F. Krieger, F. Hirner, A. C. Mert, and S. Sinha Roy, "Openntt-an automated toolchain for compiling high-performance ntt accelerators in fhe," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[24] D. Kumarathunga, Q. Hu, and Z. Fang, "Autontt: Automatic architecture design and exploration for number theoretic transform acceleration on fpgas," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2025, pp. 1–9.

[25] A. C. Mert, S. Kwon, Y. Shin, D. Yoo, Y. Lee, S. S. Roy *et al.*, "Medha: Microcoded hardware accelerator for computing on encrypted data," *arXiv preprint arXiv:2210.05476*, 2022.

[26] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2019, pp. 253–260.

[27] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[28] 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.

[29] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[30] P. N. Swarztrauber, "Multiprocessor ffts," *Parallel computing*, vol. 5, no. 1-2, pp. 197–210, 1987.

[31] C. Wang and M. Gao, "SAM: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*. IEEE, 2023, pp. 1–9.

[32] Xilinx, "Vitis unified software platform," 2025, last accessed Oct 1, 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html#development

[33] Y. Yang, R. Kannan, and V. K. Prasanna, "High throughput matrix transposition on hbm-enabled fpgas," in *33rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2025, Fayetteville, AR, USA, May 4-7, 2025*. IEEE, 2025, pp. 38–46.

[34] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: a framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 30–39.

[35] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 870–881.

[36] F. Zheng, G. Fan, W. Tang, Y. Song, T. Zhou, Y. Zhao, J. Dong, J. Lin, S. Yan, and J. Jing, "Gif-fhe: A comprehensive implementation and evaluation of gpu-accelerated fhe with integer and floating-point computing power," *IEEE Transactions on Parallel and Distributed Systems*, 2025.