Tensor-Product-Based Accelerator for Area-efficient and Scalable Number Theoretic Transform

Yuying Zhang*, Sarveswara Reddy Sathi*, Zili Kou*, Sharad Sinha†, Wei Zhang*
* Hong Kong University of Science and Technology, Hong Kong, {yzhangnf, ssreddy, zkou, weizhang}@ust.hk
† Indian Institute of Technology (IIT) Goa, India, sharad_sinha@ieee.org

Abstract—**Fully Homomorphic Encryption (FHE), which enables arbitrary computation to be performed directly on encrypted data, is becoming promising for privacy-oriented applications, paving the way for widespread adoption of cloud computing with ideal security. The challenge for FHE lies in the speed-optimized and area-optimized implementation of Number Theoretic Transform (NTT), which is the most computation-intensive primitive in FHE. Moreover, most existing works concentrate on NTT implementations with small moduli and limited levels of parallelism. The NTT designs for a wider range of parameters with high scalability, however, are not fully developed.**

This paper proposes an FPGA-based hardware accelerator for NTT with high speed and area efficiency. A novel algorithmic implementation of NTT modeled on tensor products is first proposed, which provides high flexibility in parameter sets and high scalability in processing elements (PEs). Different levels of parallelism are then explored to adapt to the trade-off between performance and area efficiency. With the help of stride permutation, a non-conflict data flow control is built to significantly simplify the memory access pattern, contributing to higher performance of NTT. Implemented on a Xilinx VIRTEX-7 platform, our RTL-based design outperforms state-of-the-art FPGA works customized for FHE by 1.21$\times \sim$ 2.73$\times$ in performance and 1.11$\times \sim$ 9.81$\times$ in area efficiency. It can achieve an enhancement of 2.49$\times$/ 1.25$\times$/ 2.53$\times$/ 2.15$\times$ on average on the resource usage of LUTs/ FFs/ BRAMs/ DSPs, respectively.

I. Introduction

Due to the growing recognition on the importance of data privacy and integrity, many privacy-preserving computing techniques have been proposed for a wide range of scenarios such as cloud computing [1], secure database search [2], and machine learning [3], among which FHE has gained broad prospects thanks to its capability of computing directly on encrypted data [4]–[6]. Despite the widely adoption in privacy-oriented applications, the efficiency of FHE is far from enough. Because of the compulsory computations on polynomials and vectors, the implementation of FHE is still 10,000$\times$ to 100,000$\times$ slower compared with computation on unencrypted data executed in an optimized software [8]. These computation-intensive operations are the fundamental computations in FHE called primitives including modular addition, multiplication, NTT and automorphism, which can form various functional units and implement homomorphic operations. Based on the fact that all FHE schemes use the same data type for ciphertexts: polynomials where each coefficient is an integer modulo a modulus [8], accelerating FHE primitives can be a feasible strategy, e.g., designing fully pipelined modular multiplications [13], vectorized automorphism unit [8], highly-parallelized NTT architectures [22] boosted by hardware, etc. Among all FHE primitives, NTT forms the main bottleneck and occupies the primary computing resources, e.g., Microsoft SEAL, an open-source library for one of the FHE schemes (CKKS), spends 54.01% of the entire time on computing NTT transformations [7]. Therefore, by accelerating NTT on hardware, one can dramatically improve the performance of FHE applications, paving the way for general and practical privacy-preserving applications. As one of the hardware platforms, FPGA offers an appealing prospect for acceleration with high parallelism and superior programmability, for which our efforts will focus on FPGA-based NTT implementation.

In addition to the performance issue, the flexibility in NTT parameters is also of great significance. Not like small sizes of polynomials ($2^8 \sim 2^{10}$) and modulus (14-bit) needed in prior NTT works for traditional cryptography applications [14]–[21], FHE requires substantially larger data to gain as much computation depth as possible with tolerable noise since FHE schemes are based on learning with errors (LWE) problems [9]. Typically, the modulus size in FHE is larger than 109-bit, and the ciphertext polynomial size should be $2^{11} \sim 2^{15}$ for practical applications [8]–[12]. Moreover, different FHE schemes apply a wide range of parameters, leading to the significance of flexibility. The large size of FHE parameters also results in the intensive occupation of hardware resources, making area efficiency a significant factor. In conclusion, an efficient hardware implementation for NTT with high area efficiency and flexibility is in great demand.

However, existing NTT implementations with high performance on FPGA mainly concentrated on the small specific modulus with fixed polynomial sizes [14]–[21]. These designs can carry out elaborate optimizations based on the unique identity of that modulus to obtain high speed. All of these works are fixed with 2 or 4 butterfly units and can not be changed. Though satisfying the requirements for some cryptography applications, they are insufficient for FHE applications due to their small size and poor flexibility. Work [22] presented two 64-PE architectures and supported large FHE parameters with high performance at the cost of considerable resources. It is applicable to any NTT-friendly prime modulus with word-level Montgomery reduction algorithm. However, the flexibility is limited because the optimization only aims at a modulus length from 20 to 32 bits. On the other hand, some state-of-the-art scalable NTT works with flexibility have restrictions on performance and area efficiency [13] [24] [28]. Due to their complex data flow between NTT stages, the physical routing when implemented on hardware can be intractable with the increment of modulus size and PEs, limiting the concurrency and extension. Also, high memory bandwidth will restrict the frequency and increase the latency significantly. Work [23] targets an area-efficient and scalable NTT design, which offers considerable advancements over the prior state-of-the-art works. However, its memory footprint needs improvements, which is a significant obstacle when this NTT design is embedded into FHE design.

To fill the research gap, we propose an area-efficient and scalable accelerator for NTT based on FPGA. In detail, our contributions in this work are summarized as follows:

1) We propose a novel model for NTT using tensor products to exploit the inherent algebraic structure for high parallelism. Based on our review, this is the first work to apply tensor products to NTT designs. Our model provides an analytic tool for various optimizations, including an improved memory access scheme of coefficients and twiddle factors at each stage.

2) We develop a simplified and non-conflict memory access pattern to lower the complexity of data flow for high scalability and area efficiency using stride permutation based on the model. We also reduce the storage of twiddle factors and keep them consistent throughout the computation by figuring out the underlying relationship between their address and value.

3) We implement a flexible and scalable NTT hardware architecture with high area efficiency on the Xilinx VIRTEX-7 FPGA platform using Verilog. Experimental results indicate the advantage of our work in performance and area efficiency. The flexibility can be shown from the adaptability in a wide range of parameter sets, while the scalability is embodied in the easy extension of the number of PEs to achieve different degrees of parallelism.

The organization of this article is as follows. Section II provides the basic knowledge about NTT operation and tensor products. Detailed formulation and hardware design are provided in Section III and Section IV, respectively. Our implementation results and comparison with prior works are shown in Section V.

II. Background

A. Number Theoretic Transform

NTT is a method to perform fast multiplication on integer polynomials, which can reduce the complexity of polynomial multiplication from n^2 to $nlogn$. Two integer polynomials are converted into the NTT domain, and then only element-wise multiplication is needed instead of polynomial multiplication. After that, INTT is used to transfer the products from the NTT domain to get the actual results of the polynomial multiplication. NTT is similar to Fast Fourier transform (FFT) but has no roundoff errors.

NTT is one of the discrete Fourier transforms defined over the quotient ring $\mathbb{Z}_q$ of the integers modulo a prime q. It maps N polynomial coefficients $(x_0, x_1, ..., x_{N-1})$ in $\mathbb{Z}_q$ into NTT domain by the form $y(m) = \sum_{n=0}^{N-1} x_n \omega_0^{mn} \; mod \; q$, where N is the transform length i.e., the number of polynomial coefficients and prime q is the modulus. ω_0 is the primitive root modulo N in $\mathbb{Z}_q$ satisfying the condition: every integer coprime to N is congruent to a power of ω_0 modulo N. The powers of ω_0 used in the transform are called twiddle factors. They have the following features: $\omega_0^N \equiv \; 1 \; mod \; q$ and for any integers a, b, if $a \equiv b \mod \; q$, then $\omega_0^a \equiv \omega_0^b \; mod \; q$. Similarly, INTT is given as $z(m) = N^{-1} \sum_{n=0}^{N-1} y_n \omega_0^{-mn} \; mod \; q$. N and the bit length of q are associated with the flexibility of NTT designs, which form the parameter set in a later discussion.

B. Residue Number System

Overwhelming computation cost will be caused by long vectors' arithmetic [8], which is the main challenge in FHE. Residue Number System (RNS) provides a feasible method for partitioning large FHE modulus into several acceptable ones. It enables representing a single polynomial with wide coefficients as multiple polynomials with narrower coefficients. The modulus Q is chosen to be the product of L smaller coprime integers, i.e. $Q = q_1 q_2 \cdot\cdot\cdot q_L$. Then a polynomial in $\mathbb{Z}_Q$ can be represented as L residue polynomials in $\mathbb{Z}_{q_1}, \mathbb{Z}_{q_2}, \cdots, \mathbb{Z}_{q_L}$. All FHE operations can be carried out under RNS representation, which can have either better or equivalent complexity than operating on one wide-coefficient polynomial [8]. RNS can also offer a level of parallelism in NTT design. Several residue polynomials can be computed in parallel in separate NTT modules, which can further exploit the inherent parallelism of FPGA. In this work, RNS is used to speed up FHE-friendly NTT operations and extend the design to much larger parameters.

C. Tensor Products and Stride Permutation

Tensor Decomposition is the scheme using a sequence of tensors as fundamental building blocks of a single complicated operation. Tensor products can offer a mathematical formulation for presenting and analyzing Digital Signal Processing (DSP) Algorithms in a unified format using matrix factorization as a form of Tensor Decomposition. They can manipulate the factorization of NTT matrices and thus provide easy conversion between vector processing and parallel processing to match specific hardware architectures with their inherent algebraic structure [25]. Tensor products can also offer substantial flexibility in terms of the degree of parallelism. $A_M \otimes B_L$ denotes the tensor product of two square matrices A and B whose sizes are $M \times M$ and $L \times L$ respectively. It means every element of the first matrix A is replaced by the product (which is also a matrix) of this element and the whole second matrix B, and finally a matrix of size ML will be obtained.

A parallel operation and vector operation using tensor products are represented as $(I_L \otimes A_M)\mathbf{x}$ and $(A_M \otimes I_L)\mathbf{x}$, respectively. I_L denotes the $L \times L$ identity matrix and A denotes a matrix of size M. $\mathbf{x}$ is the vector of size ML. Representation $I_L \otimes A_M$ corresponds to parallel processing, which means there are L separate processors used for the same computation of consecutive segments of $\mathbf{x}$ in parallel, while representation $A_M \otimes I_L$ corresponds to vector processing that can perform a vector computation of A_M directly on the sub-vectors of $\mathbf{x}$. Due to the inherent algebraic structure of $I_L \otimes A_M$ and

$A_M \otimes I_L$, these operations are more suitable for parallel or vector processors respectively. For parallel processing, when less than L separate processors are available, the following identity is used:

$$I_L \otimes A_M = I_{L_1} \otimes (I_{L_2} \otimes A_M),\ L = L_1 L_2 \qquad (1)$$

The operation $I_{L_2} \otimes A_M$ is assigned to L_1 processors to perform the computation separately. This factorization provides a theoretical formulation for the scalability of parallelism.

Stride permutation, a type of permutation, is denoted as $P(N, M)$. It represents the N-point stride M permutation and $P(N, M)\mathbf{x}$ represents the strides through vector $\mathbf{x}$. This matrix $P(N, M)$ can be decomposed using the following rule:

$$P(N, M) = P(N, M_1)P(N, M_2),\ M = M_1 M_2 \qquad (2)$$

Stride permutation provides a mathematical tool to interchange parallel processing and vector processing, giving the flexibility to partially parallelize or vectorize a tensor product computation [25]. The bridge between parallel processing and vector processing provided by stride permutation is formulated as commutation theorem:

$$P(N, M)(I_L \otimes A_M) = (A_M \otimes I_L)P(N, M),\ N = ML \quad (3)$$

The granularity of parallelism and vectorization can be controlled to different levels using this theorem. Therefore, algorithms using tensor products can adapt to one certain architecture and take full advantage of its properties.

Stride permutation also provides a mathematical language to control the required data flow in NTT designs. All NTT algorithms contain multiple stages. A series of operations are performed at each stage, after which data movement is needed to get the correct data order for the next stage. This memory access pattern can be extremely complex when N is large, holding back the computation speed. Stride permutation can reduce the complexity by providing the processors with automatic addressing, a natural way of representing data movement. The process will be explained in detail in the next section.

III. Formulation and Algorithm Optimization

A. Formulation

Cooley-Tukey FFT algorithm using tensor products has been developed for a long time [25]. In this work, we generalized it to NTT and modeled it using tensor products. The NTT operation in our work is defined as follows:

$$NTT(\mathbf{x}) = F(2^k)\mathbf{x}\ mod\ q \qquad (4)$$

where $k = log_2 N$ and $\mathbf{x}$ is the vector of input polynomial with N coefficients. NTT matrix $F(2^k)$ is a square matrix of order 2^k, which is used to transform the polynomial of size N into NTT domain. It can be decomposed into a smaller matrix with different sizes to get different levels of parallelism or vectorization. Our target platform is FPGA. To take full advantage of its high-parallelism property, full parallel optimization is utilized to factorize the whole NTT matrix into $F(2)$ of order 2. The function of $F(2)$ can be represented by the butterfly operation. The factorization is transferred from FFT [25] and shown in the following equation:

$$F(2^k) = [\prod_{l=1}^{k}(I_{2^{l-1}} \otimes F(2) \otimes I_{2^{k-l}})(I_{2^{l-1}} \otimes T_{2^{k-l}}(2^{k-l+1}))]Q(2^k) \qquad (5)$$

where k stages are needed. l denotes the variable showing the current stage number. $Q(2^k)$ is the matrix used to complete the bit reversal operation on vector $\mathbf{x}$ that can be implemented in the preprocessing. We use $\bar{\mathbf{x}}$ to denote the vector after bit reversal. Twiddle factor (TF) matrix $T_{2^{k-l}}(2^{k-l+1})$ is a diagonal matrix specified by the following rule [25]. We define the diagonal matrix $D_R(S)$ as:

$$D_R(S) = diag(1,\ \omega,\ \omega^2,\ \cdots,\ \omega^{R-1}) \qquad (6)$$

where ω is the primitive root modulo S. Then the expression of $T_R(S)$ is obtained from $D_R(S)$:

$$T_R(S) = diag(I_R,\ D_R(S),\ D_R(S)^2,\ \cdots,\ D_R(S)^{G-1}) \quad (7)$$

where $S = RG$. $T_R(S)$ can be viewed as G diagonal blocks of size R. Based on the rule, all the elements in this matrix are different powers of ω, varying from stage to stage. Therefore, $(I_{2^{l-1}} \otimes T_{2^{k-l}}(2^{k-l+1}))$ is the matrix with 2^{l-1} twiddle factor matrices. It is denoted as $Z_0(l)$ where l suggests that this matrix changed with the stage.

B. Algorithm Optimization

At each stage, the main operator $(I_{2^{l-1}} \otimes F(2) \otimes I_{2^{k-l}})$ is of mixed type involving 2^{l-1} copies of the vector operation $F(2) \otimes I_{2^{k-l}}$, which is called multidimensional tensor products. The factorization of multidimensional tensor products can be expressed as follows with the help of stride permutation [25]:

$$I_{N_1} \otimes A_{N_2} \otimes I_{N_3} = P(N, N_1 N_2)(I_{N_1 N_3} \otimes A_{N_2})P(N, N_3) \quad (8)$$

where $N = N_1 N_2 N_3$. This can convert the multidimensional tensor products into full parallelization, which is plugged into equation (5):

$$NTT(\mathbf{x}) = [\prod_{l=1}^{k} P(2^k, 2^l)(I_{2^{k-1}} \otimes F(2))P(2^k, 2^{k-l})Z_0(l)]\bar{\mathbf{x}}\ mod\ q \qquad (9)$$

The corresponding data flow of this deduced equation is shown in Fig.1(a) in the case of $N = 8$. There are k stages. The operation at each stage is divided into three parts naturally, including Modular Multiplication (MM), Tensor Products (TP) and readdressing (RA) represented by stride permutation matrices. Wr means the write operation after these calculations. This data flow suggests that two readdressing operations are required at each stage, which will cause much overhead in memory transfer of coefficients especially when N is large. Optimizations can be conducted by exploiting the properties of stride permutation. Specifically, equation (9) is unfolded along the stages first and the following expression is obtained:

$$\begin{aligned} NTT(\mathbf{x}) = &[P(2^k, 2)(I_{2^{k-1}} \otimes F(2))P(2^k, 2^{k-1})Z_0(1)] \cdots \\ &[P(2^k, 2^{(l-1)})(I_{2^{k-1}} \otimes F(2))P(2^k, 2^{k-(l-1)})Z_0(l-1)] \\ &[P(2^k, 2^l)(I_{2^{k-1}} \otimes F(2))P(2^k, 2^{k-l})Z_0(l)] \cdots \\ &[P(2^k, 2^k)(I_{2^{k-1}} \otimes F(2))P(2^k, 2^0)Z_0(k)]\bar{\mathbf{x}}\ mod\ q \end{aligned} \qquad (10)$$

The first stride permutation $P(2^k, 2^l)$ at each stage l can be factorized into two stride permutations based on equation (2), i.e., $P(2^k, 2^l) = P(2^k, 2^{l-1})P(2^k, 2)$. After the partition, the former can be a part of the last stage $(l-1)$ while the latter can stay at the present stage (l). This is specified in the following equation:

$$
\begin{aligned}
&NTT(\mathbf{x}) = \\
&[P(2^k,2)(I_{2^{k-1}} \otimes F(2))P(2^k,2^{k-1})Z_0(1)P(2^k,2)] \cdots \\
&[P(2^k,2)(I_{2^{k-1}} \otimes F(2))P(2^k,2^{k-(l-1)})Z_0(l-1)P(2^k,2^{l-1})] \\
&[P(2^k,2)(I_{2^{k-1}} \otimes F(2))P(2^k,2^{k-l})Z_0(l)P(2^k,2^l)] \cdots \\
&[P(2^k,2)(I_{2^{k-1}} \otimes F(2))P(2^k,2^0)Z_0(k)P(2^k,2^k)]\bar{\mathbf{x}} \ mod \ q
\end{aligned} \tag{11}
$$

This leads to a unified formulation at each stage. The novel theoretical formulation is presented in equation (12):

$$
NTT(\mathbf{x}) = [\prod_{l=1}^{k} P(2^k,2)(I_{2^{k-1}} \otimes F(2))Z_{2^k,l}]Q(2^k)\mathbf{x} \ mod \ q \tag{12}
$$

where diagonal matrix $Z_{2^k,l}$ satisfies

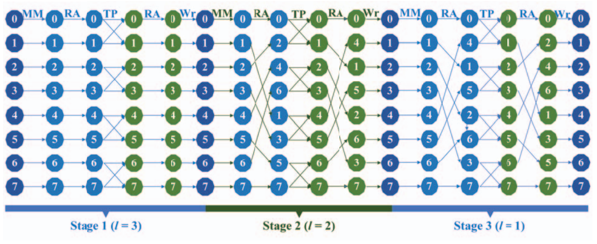
$$
Z_{2^k,l} = P(2^k,2^{k-l})(I_{2^{l-1}} \otimes T_{2^{k-l}}(2^{k-l+1}))P(2^k,2^l) \tag{13}
$$

Therefore, the complexity of the memory access pattern of the coefficients is converted into the choice of twiddle factors at each stage, which can be scheduled before NTT operation starts. $Z_{2^k,l}$ at stage l denotes how to implement this schedule which will be discussed later.

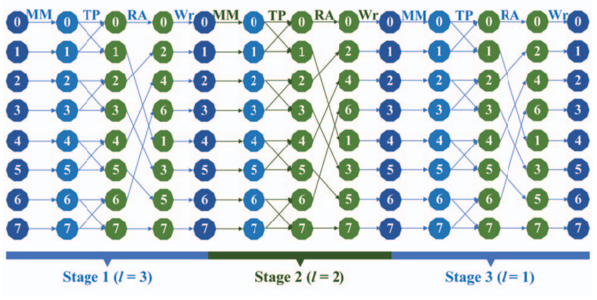
When calculating equation (12) from right to left, a matrix-vector product can always be obtained, which will be specified as different operations. And thus it is in the first stage when $l = k$. The complete data flow of the NTT operation after optimization is shown in Fig.1(b) in the case of $N = 8$. Each stage contains three matrix operations including $Z_{2^k,l}\mathbf{x}$, $(I_{2^{k-1}} \otimes F(2))\mathbf{u}$ and $P(2^k,2)\mathbf{v}$, corresponding to the modules of Modular Multiplication (MM), Tensor Products (TP) and readdressing (RA) in the hardware design, respectively. Vector $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{v}$ are the vectors obtained from the former matrix operation. For the first stage, $\mathbf{x}$ is the original input vector after bit reversal operation $Q(2^k)$.

$Z_{2^k,l}\mathbf{x}$ performs modular multiplication of every coefficients in vector $\mathbf{x}$ and the corresponding twiddle factor in Z. The matching twiddle factor is obtained from TF matrix $T_{2^{k-l}}(2^{k-l+1})$ after both tensor product and stride permutation operations. This mathematical equation (13) is converted into a practical format for FPGA, comprising one part of the memory access pattern. The details will be provided in the next section. For TP part $(I_{2^{k-1}} \otimes F(2))\mathbf{u}$, all the multidimensional tensor products are transferred into parallel operations without vector processing. It means the N-point NTT matrix is decomposed into $N/2$ 2-point NTT matrix, all of which are independent and thus can be implemented in parallel using separate PEs. Modular addition and subtraction are employed to implement each TP operation. RA implementation is also a part of the memory access pattern. It corresponds to the stride permutation operation $P(2^k,2)\mathbf{u}$, which is the same regardless of stages, making it clear to transfer data between stages.

Algorithm 1 describes the FPGA-friendly implementation details of NTT operation modeled on tensor products based on our optimized theoretical equation. The vector $\mathbf{x}$ is the original input after bit reversal, and the output $NTT(\mathbf{x})$ is the converted form of the original vector in NTT domain. MM, TP and RA are expressed into achievable forms on FPGA in Algorithm 1 and will be pipelined to improve the performance in later implementation. It should be noted that the second loop in Algorithm 1 is performed in parallel without any delay. This is because multiple processing elements (PEs) are utilized



(a) Data Flow of Originally Deduced NTT Operation



(b) Data Flow of Optimized NTT Operation

Fig. 1. Data Flow of NTT Operation when $N = 8$

Algorithm 1: NTT modeled on Tensor Products

```
Input: $\mathbf{x}(x) \in Z[x]/(x^N + 1)$
Input: Transform Length $N$, Modulus $q$
Input: Twiddle Factor $\omega_0$, PE number $\alpha$
Output: NTT($\mathbf{x}$) $\in Z[x]/(x^N + 1)$
$k = log_2 N$;
$\beta = (2^{k-1})/\alpha$;
for $l$ from $k$ by $-1$ to $1$ do
    for $i$ from $0$ by $1$ to $(\alpha - 1)$ do
        for $j$ from $0$ by $1$ to $(\beta - 1)$ do
            $\delta = i \cdot \beta + j$
            $\hat{\omega} = \omega_0^{\lfloor \frac{2\delta+1}{2^l} \rfloor 2^{l-1}}$
            $\mathbf{v}[\delta] = \mathbf{x}[2\delta] + \mathbf{x}[2\delta+1] \cdot \hat{\omega} \ mod \ q$
            $\mathbf{v}[\delta + 2^{k-1}] = \mathbf{x}[2\delta] - \mathbf{x}[2\delta+1] \cdot \hat{\omega} \ mod \ q$
        end
    end
    $\mathbf{x} = \mathbf{v}$
end
return $\mathbf{x}$
```

for the same operations simultaneously. Denote the number of PEs as α. Then the TP part $(I_{2^{k-1}} \otimes F(2))\mathbf{u}$ in equation (12) can be decomposed into $(I_\alpha \otimes (I_\beta \otimes F(2)))\mathbf{u}$, i.e. α PEs are performing a smaller tensor product operation $(I_\beta \otimes F(2))\mathbf{u}$ and corresponding modular multiplications at the same time. In this way, we naturally control the granularity of the parallel computation in this algorithm by changing α and fitting the computation to the number of available PEs.

IV. Hardware Design

A. Memory Access Pattern

Managing the complex data movement presents a substantial problem for NTT hardware designs. Memory access pattern varies in different NTT algorithms and can be very challenging. The situation is worse when multiple PEs are considered to provide high scalability since an additional constraint is needed to decide which PE corresponds to the address of a certain coefficient or twiddle factor apart from its order in one stage. Therefore, it is necessary to simplify and manage the data movement in NTT implementation. The management scheme contains two patterns: control the read or written address of the coefficients required at each stage and the address of twiddle factors needed in the MM Module.

Under the novel model proposed by this paper, the complexity of these two patterns can be converted to each other. The movement of the coefficients has higher overhead due to the memory transfers in both writing and reading processes while the pattern of twiddle factors is simply required in the MM module with no need for changing their places in memory at different stages. Thus the above derivation from equation (5) to (12) in Section III is fulfilled to keep memory transfers to a minimum. A detailed analysis of their implementation is shown in this section.

The data movement of the coefficients is called readdressing. Readdressing is required to guarantee that the written or read address for the stage is correct. With the help of stride permutation factorization from equation (5) to equation (12), there is no variable l in matrix $P(2^k, 2)$. Thus, the fixed readdressing pattern is obtained in Fig.1(b), which will not change with stages. This means the written address of the output data at the end of each stage can be confirmed once N and PE number are determined. Therefore, the complex memory access scheme of the coefficients is simplified by the unchangeable readdressing pattern at each stage which is beneficial to get the maximal degree of parallelism.

Implementing the automatic choice of twiddle factors for each coefficient in every MM operation is a non-trivial problem because it is quite different at various stages. It can be scheduled before NTT operation starts based on the analysis of equation (13). This equation provides a clear mathematical formulation on how to implement this addressing pattern of twiddle factors for MM Module. When a premultiplied stride permutation matrix times a diagonal matrix, the permutation is performed on the rows of that matrix, i.e. the row order of the matrix will be exchanged based on the stride permutation. In the same fashion, a postmultiplied stride permutation matrix can change the column order of the matrix. These two processes are shown in Fig.2.

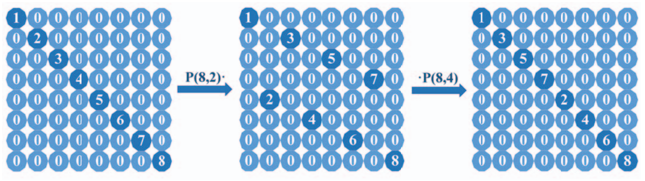


Fig. 2. One Example of Premultiplication and Postmultiplication by Stride Permutation on a Matrix

It is noticed that these two stride permutation matrices in equation (13) have the feature: $P(2^k, 2^{k-l})P(2^k, 2^l) = I_{2^k}$. Therefore, after the premultiplication and postmultiplication by stride permutation, the matrix $Z_0(l)$ changes back to a diagonal matrix with a different order of diagonal elements. Fig.2 also provides an example for this process. Define the vector composed of diagonal elements in $Z_0(l)$ as $\mathbf{z_0}$. Then the elements in $P(2^k, 2^{k-l})\mathbf{z_0}$ are exactly the diagonal elements in the obtained matrix $Z_{2^k,l}$. According to the construction rule of $Z_0(l)$ from equation (6) and (7), $\mathbf{z_0}$ is expressed as:

$$\mathbf{z_0} = (\underbrace{\overbrace{\underbrace{1,1,\cdots,1}_{2^{k-l}},\underbrace{1,\omega^1,\omega^2,\cdots,\omega^{2^{k-l}-1}}_{2^{k-l}}}^{T_{2^{k-l}}(2^{k-l+1})},\cdots)}_{2^{l-1}} \qquad (14)$$

where ω denotes the primitive root modulo 2^{k-l+1}. In this expression, $T_{2^{k-l}}(2^{k-l+1})$ points out the elements in the twiddle factor matrix, while the values under the expression show that 2^{k-l} elements in each group or 2^{l-1} twiddle factor matrices are required.

After simplifying equation (13) as above, further exploration is performed on $P(2^k, 2^{k-l})\mathbf{z_0}$. This expression means vector $\mathbf{z_0}$ is segmented into 2^l groups of size 2^{k-l} and every time one element is taken out from each group in order to get a new permutation of the whole vector until there are no elements in any group. Then the elements in the diagonal of matrix Z are obtained:

$$\mathbf{z} = \underbrace{(\underbrace{1,1,\cdots,1}_{2^l},\underbrace{1,\omega^1,\cdots,1,\omega^1}_{2^l},\cdots,\underbrace{1,\omega^{2^{k-l}-1},\cdots,1,\omega^{2^{k-l}-1}}_{2^l})}_{2^{k-l}} \qquad (15)$$

Based on the deduction of equation (15), we propose a novel method to explore the inherent relationship between the value and address of the twiddle factors and eliminate duplication. ω_0 denote the primitive root modulo 2^k, which is the known parameter in the preprocessing. The relationship between ω_0 and ω can be expressed as $\omega \equiv \omega_0^{2^{l-1}} \, mod \; q$ according to the property of twiddle factors [25]. It means that all ω in twiddle factor matrix T can be replaced by ω_0. Therefore, only the power of ω_0 needs to be explored to get the final address of a specific twiddle factor determined by $Z_{2^k,l}$. Since all the powers of ω_0 are less than 2^{k-1}, only $N/2$ different powers of ω_0 are required to be pre-loaded in order, highly reducing redundancies in the storage of twiddle factors. Then the complexity of choosing the correct twiddle factor in MM module is converted to the pattern of finding the corresponding address of pre-loaded twiddle factors. The expression of $\mathbf{z}$ yields the control logic (i.e. the address) of twiddle factors required at each stage for each coefficient. Denote the address as Z_r and the current stage number is l. Based on equation (15), when the order of the coefficient is even, $Z_r = 0$ which means the relevant twiddle factor is 1; when the order of the coefficient is odd which is denoted as t, $Z_r = \lfloor \frac{t}{2^l} \rfloor 2^{l-1}$. Thus, the memory access pattern for twiddle factors is simply described by Z_r, which is suitable for expansible designs.

In conclusion, simplified and non-conflict memory access patterns for coefficients and twiddle factors are developed respectively based on the model, lowering the complexity of data flow. As a result, the design is more suitable for extension and high parallelism with better area efficiency compared with traditional NTT designs [13] [22].

B. *PE Architecture with Flexibility and Scalability*

The whole architecture of our hardware design is shown in Fig.3, taking 4 PEs as an example. The correspondence between the theoretical formulations and their hardware implementation is presented in Table I. The architecture of one PE is shown in Fig.3, which implements the main operations in the NTT module. A PE contains a pair of multiplexers, several shift registers, one modular multiplication, and one tensor product operator. Only one modular multiplication is required in each PE to get one of the inputs for the tensor product because of our simplified memory access pattern of twiddle factors. For the other input, the shift registers are used to implement the time synchronization. Tensor product operation performs modular addition and subtraction at the same time.

Among these modules, the implementation of modular multiplication is relatively essential, which has a significant influence on the flexibility and frequency of the design. We selected Montgomery modular reduction algorithm along with the integer multiplication for modular multiplication design. Specifically, integer multiplication uses DSPs and is fully pipelined. Prior work [22] leveraged the property of NTT-friendly modulus $q \equiv 1 \ (mod \ 2N)$ in Montgomery modular reduction algorithm, avoiding one multiplication with high bit width every step in the traditional Montgomery algorithm. Thus it won't reduce the frequency. In most other NTT hardware designs, special or fixed parameters are used, so the design is not flexible and only adapts to specific applications. However, our PE hardware is applicable to different parameter sets since our tensor product operation is the fundamental unit suitable for any $N = 2^k$ and modular multiplication design is flexible for multiple choices of modulus q.

Define the number of PEs as α. To avoid read/write conflict and fully pipeline, two BRAM groups named `DATA_BRAMs` and `MID_DATA_BRAMs` are instantiated. They respectively contain α BRAMs of size N/α shown in Fig. 3. Multiplexers are used to decide the group and the exact BRAM in that group when reading based on the PE number and the stage number. Also, $\alpha/2$ twiddle factor BRAMs `TF_BRAMs` of size $N/2$ are needed to match the requirement of twiddle factors from multiple PEs. N input data and $N/2$ twiddle factors are loaded into `DATA_BRAMs` and `TF_BRAMs` respectively before the NTT operation starts. After a series of operations of each stage, the outputs should be written into the correct BRAM group. The judgment is made by the parity of k and current stage number l. They are denoted by k_parity and l_parity. If they are the same, the outputs of that stage will be stored into the corresponding group `DATA_BRAMs`. Otherwise, group `MID_DATA_BRAMs` will be chosen.

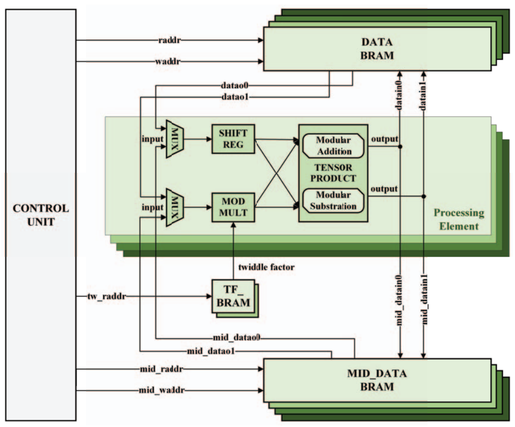


Fig. 3. Architecture of the Complete Hardware Design

TABLE I
CORRESPONDENCE BETWEEN THE THEORETICAL FORMULATIONS AND THEIR HARDWARE IMPLEMENTATION

Theoretical Formulations	$I_{2^{k-1}} \otimes F(2)$	$Z_{2^k,l}\mathbf{x}$	$P(2^k, 2)$ & $Z_{2^k,l}$
Hardware Implementation	Tensor Product	Modular Multiplication	Control Unit

In addition to determining the memory group, a clear picture of managing the data flow between stages among BRAMs is implemented by the control unit. The details are not shown in Fig.3 to prevent clutter. This global control unit can provide the addresses of each read or written coefficient and twiddle factors to govern the data flow, which implements the function of the memory access pattern introduced in the previous subsection. Because of our simplified memory access pattern as presented in the last subsection, control unit is not as complicated as other works, especially when routing with multiple PEs. Therefore, our architecture can be easily extended for multiple PEs.

Also, our hardware design can naturally support INTT with high flexibility and scalability. INTT requires an additional operation to multiply N^{-1} after all the stages finish compared with NTT. This operation is implemented by the modular multiplication module with no need for additional resources.

In summary, our NTT implementation is flexible and scalable, which can support a wide range of parameters and satisfy different demands in NTT operations, including high performance, low area, and performance-area balance by changing the number of PEs.

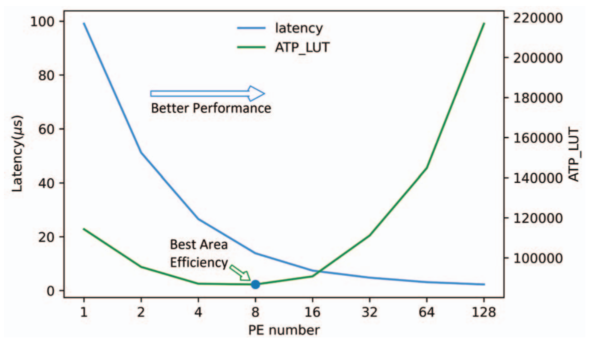


Fig. 4. Performance and Area Efficiency of Different PE Numbers on FPGA

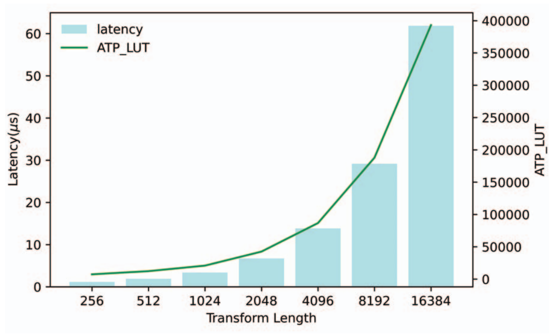


Fig. 5. Performance and Area Efficiency of Different Transform Lengths

C. Design-Space Exploration

After the complete hardware design, we present the design-space exploration in terms of the degree of parallelism (i.e. PE number) in one NTT operation to show the scalability of our work. Both latency and resource usage are crucial to evaluate NTT designs in terms of performance and area efficiency. Thus we denote the latency as the time required for one NTT operation and define ATP as the area (LUT usage) time (latency) products to quantify them, respectively. Latency reflects the throughput of NTT operations while ATP can mirror the efficiency of the resources used.

The number of PEs i.e. α reflects the degree of parallelism in one NTT operation and is associated with the performance and area efficiency of NTT, which can be a trade-off in different scenarios. Therefore, design-space exploration can be done in terms of PE number achieved by our scalable NTT design. The implementation results of latency and ATP with various PE numbers are demonstrated in Fig.4. The exploration is carried out on different parameter sets and they all share similar properties that will be shown later. So for simplification, only the results of one parameter set ($N = 4096$, $q = 32$-bit) are shown. The latency is declining along with the increase of PE number since the parallel computation leads to better performance. Despite the increment of resources all along, ATP decreases when PE number is less than 8 and grows quickly after that, suggesting the best area efficiency appears at $\alpha = 8$. This serves as a guide on how to determine the most suitable PE number for various requirements. If high performance is demanded, 32 PEs or even 64 PEs can be chosen regardless of resource usage. However, instantiating 128 PEs is not a good idea since its performance increase is not comparable with the additional resource usage i.e. the area efficiency is not acceptable from Fig.4. Typically, another consideration is how to get balanced performance and resource usage, which is relevant to area efficiency. When $\alpha = 8$, ATP is the lowest and thus best area efficiency is obtained. It is chosen in this situation due to a much better performance compared with $\alpha = 4$, which has a similar ATP. Therefore, after design-space exploration, PE number is explored to achieve a good tradeoff, providing a foundation for later discussion.

V. Evaluation

Our hardware design for both NTT and INTT is written in Verilog RTL, which is then synthesized and implemented with the default strategy using Xilinx Vivado 2021.2 on the Xilinx VIRTEX-7 FPGA platform (xc7vx690tffg1761-2) including 433200 LUTs, 866400 FFs, 1470 BRAMs and 3600 DSPs.

It is meaningful to compare the NTT implementation only when the parameters are the same including the transform length N and the bit length of the modulus q. Because of our adaptable design, we can easily modify N and q to obtain the corresponding results. Therefore, we select the parameters in the state-of-the-art works based on FPGA to make the comparison convincing.

The evaluation of our work is divided into three parts. Firstly, the implementation results in terms of different transform lengths are shown to indicate the flexibility of our work. The comparison with the state-of-the-art works for FHE applications is then analyzed in detail. Meanwhile, the best PE number is chosen to match different scenarios based on the above design-space exploration. After that, a comparison of small parameters is also demonstrated to show the versatility of our work.

A. Versatility in terms of Transform Lengths

Fig. 5 shows the performance and area efficiency when the transform length N is assigned to various values from 256 to 16384 with $q = 32$-bit and 8 PEs. This wide range of N covers the required parameters from traditional cryptography applications to FHE, suggesting the versatility of our work. The latency increases as N becomes larger shown in this figure, while the resource usage required is quite similar regardless of N according to our results. Thus, the growth trend of ATP is the same as latency. Also, the operating frequency of our design won't change with N if the bit length of q and PE number stay the same, which indicates the adaptability of our work when applied to large parameter sets.

B. Comparison to Works Customized for FHE

Specifically, the comparisons between our solution and existing works for FHE are based on three commonly-used sets of parameters shown in table II. Set A and B can be the fundamental modules to implement Set C or even larger parameter sets thanks to RNS. Table III demonstrates a thorough

TABLE II
FHE-FRIENDLY PARAMETER SETS

	N	q
Set A	4096	32-bit
Set B	4096	60-bit
Set C	4096	180-bit

comparison between prior works and our work. Four labels are attached to these works: scalable designs with different degrees of parallelism, designs with fixed parallelism, flexible designs for various parameter sets, and fixed and optimized designs for one or few specific moduli. The normalized latency and ATP of prior works compared with our design are shown when the corresponding results of our design are set to 1. These normalized results are exactly the improvements of our work in latency and ATP. Similarly, the normalized resource usage including LUT, FF, BRAM and DSP is also demonstrated under the original results in Table III, which makes our improvement in resource reduction explicit.

For Set A, we first compare our design with the RTL-based work aiming at scalability [23]. Work [23] employed a divided NTT/INTT design, which requires redundant resources. Our unified and reconfigurable NTT/INTT design can significantly reduce the usage of LUTs and BRAMs. The latency of our design is slightly worse because of 30% fewer DSPs than their work. Even so, our area efficiency is improved by 1.99×. Our work is also compared with two highly parallelized NTT hardware architectures with 64 PEs including the iterative and the four-step CooleyTukey algorithms (ITER and CT are used to represent them in Table III) [22]. The results show that our latency is higher. However, it is worth noting that they use much more DSPs than ours and have no scalability. Despite that, our area efficiency is still better. The improvement of resource usage can be up to 3.44×, 3.39× and 4.25× in terms of LUTs, BRAMs and DSPs, respectively.

For Set B, we compare our design with a state-of-the-art scalable design on FPGA [13]. Work [13] employed iterative NTT algorithm which is an in-place algorithm with a data movement scheme of high complexity. 8 PEs and 32 PEs are chosen in table III to obtain balanced and performance-optimized designs, respectively. Because of our simplified memory access pattern, higher frequency and lower resource usage including LUTs and BRAMs can be obtained. Therefore, our work can achieve 1.65× to 1.96× improvement in latency and 3.74× to 3.86× improvement in ATP, which suggests that our design has the enhancement in both performance and area efficiency regardless of the change of PE numbers. Our LUT usage save is more notable as the degree of parallelism increases, indicating our design's benefit for high-parallelism scenarios.

For Set C, the extended version of our work is carried out to adapt to the large parameter based on RNS. Work [27] applied algorithmic optimization techniques on NTT operation and block-level pipeline strategies to increase the throughput. Algorithm-level parallelism offered by RNS was employed to reduce latency. However, our work can still achieve an improvement of 2.74× in performance although 32 fewer DSPs are used. Moreover, due to the complicated memory access pattern of work [27], some strategies were used to eliminate conflicts at the cost of resources and thus the area was not optimized. Our advantage in resource usage is noticeable from Table III. Another comparison for Set C is also from work [22]. In this case, the number of our extended NTT modules will be adjusted in addition to the PE number to accommodate various circumstances. We set up $\alpha = 32$ in our design to achieve comparable performance. To compare with ITER algorithm implementation in work [22], our work is extended with three same NTT modules to balance performance and area. Although our work has slightly higher latency due to much fewer DSPs, the area efficiency is 1.57× higher than work [22]. Meanwhile, since CT algorithm implementation in [22] is concentrated on high performance, we adjust our design to six same NTT modules and achieve 1.21× and 1.37× performance and area efficiency enhancement, respectively. Moreover, all kinds of resources required in our work are lower than work [22], indicating a better area optimization of our work.

C. Comparison to Works with Small Parameters

When compared with existing state-of-the-art works designed and optimized for a specific modulus q in a specific application, we adjust our PE number to match the parallelism, and the degree of parallelism in each work is presented to clarify the comparison. The comparison results are shown in Table IV. For parameter set 1024, 14-bit, prior works employed 4 butterfly units to decrease latency which is fixed and cannot be altered in works [14] [19] [20] [21]. Our latency and ATP improvement can be up to 3.00× and 6.69× respectively. However, when compared with work [19], [20], our latency and ATP are slightly higher respectively, which is mainly because they only focus on one specific modulus and make careful optimization based on its special property. Even so, we can still achieve either lower latency or better area efficiency. When compared with prior RTL-based scalable designs on small parameters [13], it is superior in both performance and area efficiency regardless of the number of PEs, with enhancements of 1.62× and 2.05× on average, respectively.

VI. CONCLUSION

We present an area-efficient and scalable NTT accelerator for multiple scenarios based on our novel NTT formulation modeled on tensor products. The memory access pattern is simplified to a large extent by the inherent algebraic structure of our model. These optimizations can lead to improved performance and area efficiency along with flexibility and scalability. The distinguished flexibility and scalability can adapt our work to various applications with different parameter sets and levels of parallelism. Later work will focus on a different mathematical formulation of NTT from this work by manipulating the factorization. This can further decrease memory utilization, which will have higher adaptability for the whole FHE implementation in future applications.

TABLE III
OUR IMPLEMENTATION RESULTS AND COMPARISON TO PRIOR FHE-FRIENDLY WORKS

Hardware Design	Platform	N	q	LUTs/FFs/BRAMs/DSPs (Normalized Area)	Frequency (MHz)	Latency (μs)	ATP_LUT	Normalized latency	Normalized ATP_LUT
This Work-8PE^{a,c}	**Virtex-7**	**4096**	**32-bit**	**6260/7132/20/56**	**235**	**13.85**	**86701**	**1×**	**1×**
[23]^{a,c}	Virtex-7	4096	32-bit	14004/8662/79/80 (2.24/1.21/3.29/1.43)	250	12.3	172279	0.89×	1.99×
This Work-32PE^{a,c}	**Virtex-7**	**4096**	**32-bit**	**46547/26937/192/448**	**184**	**3.11**	**144761**	**1×**	**1×**
[22]-ITER^{b,c}	Virtex-7	4096	32-bit	70000/-/129/599 (3.01/-/1.34/2.67)	200	2.3	161000	0.74×	1.11×
[22]-CT^{b,c}	Virtex-7	4096	32-bit	80000/-/325.5/952 (3.44/-/3.39/4.25)	200	1.75	140000	0.37×	1.26×
This Work-8PE^{a,c}	**Virtex-7**	**4096**	**60-bit**	**11790/13336/48/112**	**244**	**13.36**	**157514**	**1×**	**1×**
[13]-8PE^{a,c}	Virtex-7	4096	60-bit	23215/-/176/248 (1.97/-/3.67/2.21)	125	26.2	608233	1.96×	3.86×
This Work-32PE^{a,c}	**Virtex-7**	**4096**	**60-bit**	**43696/50322/192/448**	**204**	**4.68**	**204498**	**1×**	**1×**
[13]-32PE^{a,c}	Virtex-7	4096	60-bit	99384/-/176/992 (2.27/-/0.92/2.21)	125	7.7	765257	1.65×	3.74×
***This Work-8PE**^{a,c}	**Virtex-7**	**4096**	**180-bit**	**17685/20004/72/168**	**244**	**26.72**	**472543**	**1×**	**1×**
[27]^{b,c}	Zynq UltraScale+	4096	180-bit	63522/25622/400/200 (3.59/1.28/5.56/1.19)	225	73	4637106	2.73×	9.81×
***This Work-32PE-1**^{a,c}	**Virtex-7**	**4096**	**180-bit**	**65544/75483/192/672**	**204**	**9.36**	**613492**	**1×**	**1×**
[22]-ITER^{b,c}	Virtex-7	4096	180-bit	140000/-/258/1198 (2.14/-/0.90/1.78)	200	6.9	966000	0.74×	1.57×
***This Work-32PE-2**^{a,c}	**Virtex-7**	**4096**	**180-bit**	**131088/150966/576/1344**	**204**	**4.68**	**613492**	**1×**	**1×**
[22]-CT^{b,c}	Virtex-7	4096	180-bit	160000/-/651/1904 (1.22/-/1.13/1.42)	200	5.25	840000	1.21×	1.37×

ATP_LUT: Area (LUT) Time (latency) Product (lower is better)
[a]: Scalable designs supporting different levels of parallelism [b]: Designs with fixed parallelism
[c]: Flexible designs for various parameter sets [d]: Designs for the specific modulus
*means the extended version of this work using RNS to compare with other designs.

TABLE IV
OUR IMPLEMENTATION RESULTS AND COMPARISON TO PRIOR WORKS WITH SMALL PARAMETERS

Design	Levels of parallelism	Platform	LUTs	BRAMs	DSPs	Frequency (MHz)	Latency (μs)	ATP_LUT	Normalized latency	Normalized ATP_LUT
					$N = 1024, q =$ 14-bit					
This Work^{a,c}	**1**	**Virtex-7**	**578**	**1.5**	**3**	**250**	**21.02**	**12150**	**1×**	**1×**
This Work^{a,c}	**4**	**Virtex-7**	**1271**	**5**	**12**	**244**	**5.81**	**7385**	**1×**	**1×**
This Work^{a,c}	**8**	**Virtex-7**	**2304**	**10**	**24**	**241**	**3.22**	**7419**	**1×**	**1×**
This Work^{a,c}	**32**	**Virtex-7**	**8834**	**40**	**96**	**225**	**1.32**	**11661**	**1×**	**1×**
[14]^{b,d}	4	Zynq-7000	2832	10	8	150	17.45	49418	3.00×	6.69×
[19]^{b,d}	4	Virtex-7	10758	-	36	302	0.85	9144	0.15×	1.24×
[20]^{b,d}	4	Artix-7	798	-	4	234	6.8	5426	1.17×	0.73×
[21]^{b,d}	4	Zynq-7000	4823	-	8	153	8.37	40369	1.44×	5.47×
[13]^{a,c}	1	Virtex-7	575	11	3	125	41.28	23736	1.96×	1.95×
[13]^{a,c}	8	Virtex-7	2584	16	24	125	5.44	13954	1.69×	1.88×
[13]^{a,c}	32	Virtex-7	17188	48	96	125	1.6	27501	1.21×	2.36×

ATP_LUT: Area (LUT) Time (latency) Product (lower is better)
[a]: Scalable designs supporting different levels of parallelism [b]: Designs with fixed parallelism
[c]: Flexible designs for various parameter sets [d]: Designs for the specific modulus

VII. Acknowledgements

This research was supported by AI Chip Center for Emerging Smart Systems (ACCESS), Hong Kong SAR and the RGC GRF grant 16213521.

References

[1] M. Hamdaqa, L. Tahvildari, “Cloud Computing Uncovered: A Research Landscape,” Advances in Computers, vol. 86, pp. 41–85, 2012.

[2] M. H. Mughees, H. Chen, and L. Ren, “Onionpir: Response efficient single-server PIR,” ACM SIGSAC Conference on Computer and Communications Security, pp. 2292–2306, 2021.

[3] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated machine learning: Concept and applications,” ACM Trans. Intell. Syst. Technol., vol. 10, no. 2, pp. 12:1–12:19, 2019.

[4] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” Cryptol. ePrint Arch., Tech. Rep. 2012/144, 2012.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in Proc. Int. Conf. Theor. Appl. Cryptol. Informat. Secur., Hong Kong, Dec. 2017, pp. 409–437.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” ACM Trans. Comput. Theory, vol. 6, pp. 1–36, Jul. 2014.

[7] “Microsoft SEAL (release 3.2),” https://github.com/Microsoft/SEAL, Feb. 2019, microsoft Research, Redmond, WA.

[8] F. Axel, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, “F1: A fast and programmable accelerator for fully homomorphic encryption,” in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, pp. 238–252, 2021.

[9] K. Sangpyo, J. Kim, M. Jaemin Kim, et al. “BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption,” ISCA, 2022.

[10] N. Samardzic, A. Feldmann, A. Krastev, et al, “Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data,” The 49th Annual International Symposium on Computer Architecture, pp. 173–187, 2022.

[11] J. Kim, G. Lee, S. Kim, G. Sohn, J. Kim, M. Rhu, and J. H. Ahn, “ARK: fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse,” CoRR, vol. abs/2205.00922, 2022.

[12] M. Riazi, K. Laine, B. Pelton, and W. Dai, “HEAX: An Architecture for Computing on Encrypted Data,” in International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, pp. 1295–1309, 2020.

[13] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas and A. Aysu, "An extensive study of flexible design methods for the Number Theoretic Transform," in IEEE Transactions on Computers, 2020.

[14] P. Kuo, Y. Chen, Y. Hsu, C. Cheng, W. Li, and B. Yang, “High performance Post-Quantum key exchange on FPGAs,” Journal of information science and engineering, vol. 38, pp. 1211–, 2022.

[15] Y. Yang, S. R. Kuppannagari, R. Kannan, V. K. Prasanna, “NTTGen: a framework for generating low latency NTT implementations on FPGA,” Proceedings of the 19th ACM International Conference on Computing Frontiers, pp. 30-39, 2022.

[16] T. Ye, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, “FPGA Acceleration of Number Theoretic Transform,” in High Performance Computing, vol. 12728, 2021.

[17] C. Li, W. Zhu, and L. Liu, “A High Speed NTT Accelerator for Lattice-Based Cryptography,” in 2021 IEEE 3rd International Conference on Communications, Information System and Computer Engineering, CISCE, pp. 85–89, 2021.

[18] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, “Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT,” IACR transactions on cryptographic hardware and embedded systems, vol. 2020, no. 2, 2020.

[19] P. Duong-Ngoc and H. Lee, "Configurable mixed-radix Number Theoretic Transform architecture for lattice-based cryptography," in IEEE Access, vol. 10, pp. 12732-12741, 2022.

[20] B. Niasar, Mojtaba, R. Azarderakhsh and M. Kermani, “High-Speed NTT-based polynomial multiplication accelerator for CRYSTALS-Kyber Post-Quantum cryptography,” in Proc. IEEE 28th Symp. Comput. Arithmetic (ARITH), p. 563, 2021.

[21] Y. Xing and S. Li, "An efficient implementation of the NewHope key exchange on FPGAs," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 67, pp. 866-878, 2020.

[22] A. C. Mert, E. Ozturk, and E. Savas, “Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme,” IEEE transactions on very large scale integration (VLSI) systems, vol. 28, pp. 353–362, 2020.

[23] Y. Su, B. Yang, C. Yang, Z. Yang and Y. Liu, "A highly unified reconfigurable multicore architecture to speed up NTT/INTT for homomorphic polynomial multiplication," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 30, pp. 993-1006, 2022.

[24] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi and A. Aysu, "A flexible and scalable NTT Hardware: Applications from homomorphically encrypted deep learning to Post-Quantum cryptography," 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 346-351, 2020.

[25] R. Tolimieri, M. An, and C. Lu, Algorithms for discrete Fourier transform and convolution. New York: Springer, 1997.

[26] S. Durrani et al., "Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles," 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2021.

[27] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren and I. Verbauwhede, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 387-398, 2019.

[28] R. Paludo and L. Sousa, "NTT Architecture for a Linux-Ready RISC-V Fully-Homomorphic Encryption Accelerator," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 69, no. 7, pp. 2669-2682, 2022.