IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 1, pp. 94–126. DOI:10.46586/tches.v2022.i1.94-126

CFNTT: Scalable Radix-2/4 NTT Multiplication Architecture with an Efficient Conflict-free Memory Mapping Scheme

Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei and Leibo Liu*

Beijing National Research Center for Information Science and Technology (BNRist).
School of Integrated Circuits, Tsinghua University, China.
cxr19@mails.tsinghua.edu.cn;{bohanyang,yinsy,wsj,liulb}@tsinghua.edu.cn
* Corresponding Author.

Abstract. Number theoretic transform (NTT) is widely utilized to speed up polynomial multiplication, which is the critical computation bottleneck in a lot of cryptographic algorithms like lattice-based post-quantum cryptography (PQC) and homomorphic encryption (HE). One of the tendency for NTT hardware architecture is to support diverse security parameters and meet resource constraints on different computing platforms.Thus flexibility and Area-Time Product (ATP) become two crucial metrics in NTT hardware design. The flexibility of NTT in terms of different vector sizes and moduli can be obtained directly. Whereas the varying strides in memory access of in-place NTT render the design for different radix and number of parallel butterfly units a tough problem. This paper proposes an efficient conflict-free memory mapping scheme that supports the configuration for both multiple parallel butterfly units and arbitrary radix of NTT. Compared to other approaches, this scheme owns broader applicability and facilitates the parallelization of non-radix-2 NTT hardware design. Based on this scheme, we propose a scalable radix-2 and radix-4 NTT multiplication architecture by algorithm-hardware co-design. A dedicated schedule method is leveraged to reduce the number of modular additions/subtractions and modular multiplications in radix-4 butterfly unit by 20% and 33%, respectively. To avoid the bit-reversed cost and save memory footprint in arbitrary radix NTT/INTT, we put forward a general method by rearranging the loop structure and reusing the twiddle factors. The hardware-level optimization is achieved by excavating the symmetric operators in radix-4 butterfly unit, which saves almost 50% hardware resources compared to a straightforward implementation. Through experimental results and theoretical analysis, we point out that the radix-4 NTT with the same number of parallel butterfly units outperforms the radix-2 NTT in terms of area-time performance in the interleaved memory system. This advantage is enlarged when increasing the number of parallel butterfly units. For example, when processing 1024 14-bit points NTT with 8 parallel butterfly units, the ATP of LUT/FF/DSP/BRAM in radix-4 NTT core is approximately $2.2\times/1.2\times/1.1\times/1.9\times$ less than that of the radix-2 NTT core on a similar FPGA platform.

Keywords: number theoretic transform, polynomial multiplication, algorithm-hardware co-design, radix-4, conflict-free memory mapping scheme

1 Introduction

In recent years, both industrial and academic community have sparked a boom in the research of post-quantum cryptography and homomorphic encryption. Since the Shor algorithm renders the traditional cryptosystems, like Rivest Shamir Adleman (RSA) and

Licensed under Creative Commons License CC-BY 4.0.
Received: 2021-07-15 Accepted: 2021-09-15 Published: 2021-11-19

Elliptic Curve Cryptography (ECC), to be cracked by quantum computer within polynomial time [Sho94], the National Institute of Standards and Technology (NIST) has initiated a contest to solicit post-quantum cryptographic algorithms around the world. In July 2020, 7 finalists and the 8 alternate candidates are announced in the third round and lattice-based schemes account for 5/7 of them. Lattice-based cryptography provides both solid security and hardware friendly way to build post-quantum cryptosystems [MAA$^+$20]. It also offers other advanced applications, such as fully homomorphic encryption [Gen09] and zero knowledge proof [LNS20]. The polynomial multiplication is the well-known computational bottleneck in lattice-based schemes, while leveraging number theoretic transform can reduce its complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ [Nic71]. As a result, works focused on the design of NTT hardware accelerator have been intensively carried out.

Although the NTT algorithm applied in lattice-based PQC and HE is slightly different, the overall hardware architecture has a lot of similarities. It is interesting to propose a NTT multiplication architecture as a building block for these schemes [CHK$^+$21] [MKO$^+$20a]. Besides, some resource-constrained scenarios focus on compact and energy-efficient design like IoT, while others pursue high throughput and better performance like 5G [XHY$^+$20]. To meet these diverse requirements, flexibility and Area-Time Product become two crucial metrics in NTT hardware design.

Related work. Plenty of works have developed optimization techniques for radix-2 NTT hardware architecture. But the design of radix-4 NTT hardware core is still rare. With regard to the algorithm level optimization, [POG15] and [ZYC$^+$20] demonstrate the iterative radix-2 NTT algorithm free from the pre-processing and post-processing. It reduces the computation complexity by merging the $2N$-th primitive roots of unity and scale factor into every stage. [XL21] and [POG15] avoid the bit-reversed cost by changing the loop structure of decimation in time (DIT) and decimation in frequency (DIF) radix-2 NTT. This method is also adopted by the reference software implementation of Kyber published by NIST. [ACC$^+$21] and [CHK$^+$21] introduce various NTT extension algorithms by applying versatile FFT tricks, enabling the software implementation of polynomial multiplication on NTT unfriendly rings. In terms of architecture level design, unlike the popular pipelined FFT architecture, such as single delay feedback (SDF) and multiple delay commutator (MDC) [TCH19] [HT96], the current NTT core mainly adopts memory-based architecture allowing for a trade-off between area and performance. [CMV$^+$15] proposes a high-speed pipelined polynomial multiplication architecture based on constant geometry radix-2 NTT. This paper points out the temporal conflicts in pipeline architecture. [FL19] presents a HE integer multiplier utilizing the negative wrapped convolution and Ping-pong radix-2 FFT algorithm. Compared to in-place NTT architecture, the aforementioned two types have a simpler memory access pattern at the price of double memory overheads. Actually, [GGMG13] and [SYJ84] indicate that radix-4 FFT architecture has advantages in high-throughput-demanding applications.

The conflict-free memory mapping scheme is a key component for parallel NTT hardware architecture. However, to our best knowledge, an efficient and general memory mapping scheme for arbitrary radix in-place NTT with parallel butterfly units is still in lack. [Joh92] presents a conflict-free address mapping method for arbitrary radix FFT. [LSW01] and [WHEW14] design a dedicated radix-2 FFT and radix-16 NTT architecture based on this method, respectively. But it is identified in [XMX17] and [ZYC$^+$20] that this scheme cannot be applied to the case when placing multiple parallel butterfly units into every stage. [ZYC$^+$20] handles this problem by reordering the last loop of NTT algorithm, but it finitely considers the case of radix-2 NTT. [XMX17] [TJS03] present a memory mapping approach in terms of radix-2^k constant geometry FFT (not in-place) configured with several parallel butterfly units. The vector radix-2 NTT core configured with 32 parallel butterfly units is proposed in [XHY$^+$20]. The memory access pattern is implemented by six types of permutation networks, which have complex control mode and consume much hardware resources. [RV08] puts forward an address mapping scheme supporting arbitrary radix in-place FFT and parallel butterfly units. But this scheme requires a lot of banks positioning at every stage as read-write buffer. The memory mapping approach proposed in [RMD+15] will vary with stages, increasing the complexity of the pipeline design. A constant geometry radix-2 NTT architecture with only two parallel butterfly units is demonstrated in [BUC19]. [FLX20] also proposes a parallel Stockham NTT hardware design. [XWXY17] derives an address mapping approach for parallel arbitrary radix MDC-memory-based FFT architecture. However, the aforementioned three cases consume double memory overheads. [MKÖ+20b] [RVM+14] utilize a single RAM and extra registers to construct the parallel radix-2 NTT address mapping scheme that leads to pipeline stalls.

Contribution. We release the design codes as open source on `https://github.com/xiang-rc/cfntt_ref`. In general, our contributions are summarized as follows:

1. We provide a detailed derivation for bit-reversed-free radix-4 NTT/INTT algorithm with low complexity over polynomial rings. To avoid the bit-reversed cost and reduce the memory footprint in arbitrary radix NTT/INTT, we put forward a general method by rearranging the loop structure and making full use of the properties of twiddle factors. The proposed new radix-4 DIT-NR NTT and DIF-RN INTT algorithms achieve nearly minimal number of modular multiplication $N \log_4 N$ and clock cycles $N/4 \log_4 N$ theoretically.

2. A scalable radix-2 and radix-4 NTT multiplication architecture is proposed in this paper. We adopt a dedicated schedule method to reduce the number of modular additions/subtractions and multiplications in radix-4 butterfly unit by 20% and 30%, respectively. By reusing the symmetric operators in radix-4 butterfly unit, we save almost 50% area overhead compared to a straightforward implementation. Through experimental results and theoretical analysis, we point out that the radix-4 NTT has an advantage over radix-2 NTT in terms of area-time performance in the interleaved memory system, which is enhanced when increasing the number of parallel butterfly units. The ATP of LUT/FF/DSP/BRAM in radix-4 NTT core is approximately $2.2\times/1.2\times/1.1\times/1.9\times$ less than that of the radix-2 NTT core on a similar FPGA platform when processing 1024 14-bit points NTT with 8 butterfly units.

3. The efficient memory access of in-place NTT is guaranteed by the devised new conflict-free mapping scheme, which supports the configuration for multiple parallel butterfly units and arbitrary radix. This scheme avoids introducing queues and pipeline stalls, thus enabling approximately 100% utilization of pipelined NTT architecture. It also paves the way for other non-radix-2 NTT hardware designs. In contrast to other approaches, this scheme covers broader applicability while still maintaining low logic and storage overhead.

2 Preliminaries

2.1 NTT-based Multiplication Algorithm

The traditional Fast Fourier Transform (FFT) applied in the digital signal process is defined on the complex field $\mathbb{C}$, which reduces the computation complexity of Discrete Fourier Transform (DFT) from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. The Number Theoretical Transform (NTT) can be regarded as DFT defined on finite field $\mathbb{Z}_q$, so that the vector multiplication can be computed without loss accuracy. The Inverse Number Theoretical Transform (INTT) can be obtained by replacing the twiddle factor ω of NTT with its inverse element ω^{-1} followed by multiplying the scale factor N^{-1}. The concrete formula is described as belows:

$$\begin{aligned} A_i = NTT(a_i) &= \sum_{j=0}^{N-1} a_j \omega_N^{ij} \bmod q \quad i = 0, 1, ..., N-1 \\ a_j = INTT(A_j) &= \frac{1}{N} \sum_{i=0}^{N-1} A_i \omega_N^{-ij} \bmod q \quad j = 0, 1, ..., N-1 \end{aligned} \tag{1}$$

NTT multiplication on finite field. In general, the prime modulus q needs to satisfy $q \equiv 1 \pmod N$, which ensures the existence of N-th primitive roots of unity ω. N is a power of 2. As a result, the multiplication of two N-term polynomials on finite field can be performed as equation 2, where the length of vector $\boldsymbol{a}$ and $\boldsymbol{b}$ is N, the zeropadding operation is to double the length of input vector by appending N zeroes at the end. Therefore, the length of vector $\boldsymbol{c}$ becomes $2N$.

$$\boldsymbol{c} = INTT(NTT(zeropadding(\boldsymbol{a})) \cdot NTT(zeropadding(\boldsymbol{b}))) \tag{2}$$

NTT multiplication over the ring. In lattice-based PQC algorithm, the polynomial multiplication over the ring $\mathbb{Z}_q[x]/\langle f(x) \rangle$ needs extra modular reduction with irreducible polynomial $f(x)$. But if $f(x) = x^N + 1$, we can use the well-known negative wrapped convolution (NWC) method to avoid doubling efforts and extra modular reduction. The NWC method can be expressed as equation 3, where ϕ is $2N$-th roots of unity and $q \equiv 1 \pmod{2N}$ at this time.

$$\begin{aligned} \text{Pre-processing}: &\ \overline{a_i} = a_i \cdot \phi_{2N}^i \quad \overline{b_i} = b_i \cdot \phi_{2N}^i \\ \text{Multiplication}: &\ \overline{c_i} = INTT(NTT(\overline{a_i}) \cdot NTT(\overline{b_i})) \\ \text{Post-processing}: &\ c_i = \overline{c_i} \cdot \phi_{2N}^{-i} \end{aligned} \tag{3}$$

The so-called pre-processing and post-processing can be eliminated by merging factors ϕ_{2N}^i (ϕ_{2N}^{-i}) into every stage of NTT [ZYC+20]. While the bit-reversed operation can be

Algorithm 1 Radix-2 Bit-reversed-free DIT NTT Algorithm

```
Input: a denotes a vector of length N, q is modulus and $\phi_{2N}$ is 2N-th primitive roots of unity.
Output: $\boldsymbol{A} = DIT\_NTT(\boldsymbol{a})$
1: $r \leftarrow 1$
2: for $p = \log_2 N - 1$ to 0 do
3:     $J \leftarrow 2^p$
4:     $\omega_m \leftarrow \phi_{2N}^{N/(2J)}$
5:     for $k = 0$ to $N/(2J) - 1$ do
6:         $\omega \leftarrow \omega_m^{\text{bit-reversed}(r)}$
7:         $r \leftarrow r + 1$
8:         for $j = 0$ to $J - 1$ do
9:             $u \leftarrow a_{2kJ+j}$
10:            $v \leftarrow a_{2kJ+j+J}$
11:            $A_{2kJ+j} \leftarrow (u + v \cdot \omega) \bmod q$
12:            $A_{2kJ+j+J} \leftarrow (u - v \cdot \omega) \bmod q$
13:        end for
14:    end for
15: end for
16: return $\boldsymbol{A}$
```

further removed by rearranging the loop structure [XL21]. The complete bit-reversed-free radix-2 iterative NTT with low complexity is presented in algorithm 1, where the twiddle factors ω can be precomputed and stored in the ROM. The operations within the innermost loop are commonly defined as the butterfly computation owing to its butterfly-like dataflow

topological structure. [ACC+21] and [CHK+21] introduce a series of NTT extension algorithms, e.g., mixed-radix, Good's trick and multiple modulus NTT, which allows the NTT multiplication to be performed on NTT-unfriendly rings. However, these novel algorithms are only implemented on the software platform Cortex-M4 and AVX2.
Radix-2^k NTT algorithm. The appearance of radix-2^2 is a milestone in FFT hardware design, which is extended to radix-2^k later [GGMG13]. The radix-4 FFT is preferable to be applied in high-throughput-demanding applications [SYJ84]. In analogy with FFT, we can also obtain the radix-2^k NTT algorithm. Actually, in [ABCG20] [BKS19] [GOPS13] [CHK+21], the radix-2^k NTT algorithm is realized by merging multiple layers of radix-2 NTT on the resource-constrained micro-controller platforms. This method is leveraged to reduce the cache loading and storing overheads. In this paper, we will provide the detailed derivation of bit-reversed-free radix-4 NTT/INTT algorithm with low computation complexity. Furthermore, we point out that its hardware advantage in terms of better area-timing trade-off is benefited from the symmetric operators and less number of banks.
Properties of twiddle factor. The twiddle factor used in NTT algorithm refers to powers of the N-th primitive roots of unity (ω_N^i). In the deduction process, we will show that the low computation complexity and minimal memory footprint of NTT algorithm are based on the unique properties of twiddle factors, which can be generalized as four items :

$$\begin{aligned}
\text{Sum property:}\quad & \sum_{j=0}^{N-1} \omega_N^{kj} = \frac{(\omega_N^k)^N - 1}{\omega_N^k - 1} = \frac{(\omega_N^N)^k - 1}{\omega_N^k - 1} = 0 \quad \text{where} \quad k \nmid N \\
\text{Binary property:}\quad & \omega_N^{k+N/2} = -\omega_N^k \\
\text{Periodicity property:}\quad & \omega_N^{k+N} = \omega_N^k \\
\text{Elimination property:}\quad & \omega_{dN}^{dk} = \omega_N^k
\end{aligned} \tag{4}$$

It is easy to see that the Binary property can be derived from Elimination property and the $2N$-th roots of unity ϕ_{2N} also has similar properties.

2.2 Scalability of NTT Architecture

2.2.1 Dimensions of Scalability

Motivation. Works in [BUC19] [XHY+20] [MKÖ+20b] consecutively propose the configurable PQC cryptosystems based on a NTT processor with different vector length and moduli. [BUC19] mainly targets energy efficiency for IoT and thus proposes a NTT core with 2 butterfly units. While [XHY+20] aims to achieve high performance for 5G and puts forward a vector NTT processor with 32 butterfly units. To support the configuration for diverse security parameters and satisfy different resource constraint of computation platform, we propose a scalable NTT multiplication architecture in this paper. Based on this design paradigm, one can generate the desired lightweight or high-performance NTT core for different application scenarios. Our scalable in-place NTT multiplication architecture considers four dimensions of scalability: length of data vector, different modulus, number of parallel butterfly units and different radix, which are denoted as N, q, d, R, respectively. While the configuration for N and q can be easily obtained, supporting different radix and multiple parallel butterfly units becomes a tough problem owing to the complex memory access in every stage. In fact, the interconnect network is an important component of NTT hardware, which guarantees the data points to be fetched and stored without conflicts so that no queues and reordering are needed. For example, [XHY+20] increases the number of parallel butterfly units to achieve high throughput in radix-2 NTT. For correct memory access, it proposes a permutation network, which needs six configuration modes and consumes more hardware resources. To reduce these overheads, we propose an efficient conflict-free memory mapping scheme. This scheme is also valid for parallel non-radix-2 NTT hardware design.

Interleaved memory system. If the radix-R in-place NTT contains d butterfly units, R × d memory access should be performed in parallel. The memory bandwidth can be increased by partitioning the memory into R × d independent banks, which is referred to an interleaved (matched) memory system [TJS03]. Distributing data points over the interleaved memory system without conflict is a more difficult and universal problem compared to that of unmatched memory system. Because multiple data points can be stored into a single address by increasing the row-bandwidth in unmatched memory system. Its storage structure can be straightforwardly obtained according to the mapping scheme of the interleaved memory system. Thus we focus on the memory mapping scheme for in-place NTT with interleaved memory system in this article.

2.2.2 Temporal and Spatial Conflicts

Temporal conflict. In pipelined NTT architecture, temporal conflict occurs when the read operation of stage k is performed before the write operation of stage $k-1$ [CMV+15]. Figure 1 provides an example of 16-point in-place radix-2 NTT for this read-after-write (RAW) conflict. At the first round of stage 1, the four data points (a_0, a_2, a_1, a_3) are fetched from the banks right at the moment when the four data points (a_0, a_1, a_2, a_3) at stage 0 are written into the banks, which results in the critical point of temporal conflict. In pipeline radix-2 in-place NTT architecture, the transitional moment between the last

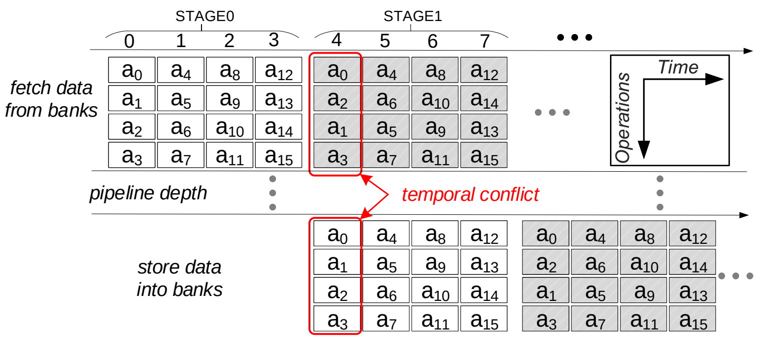
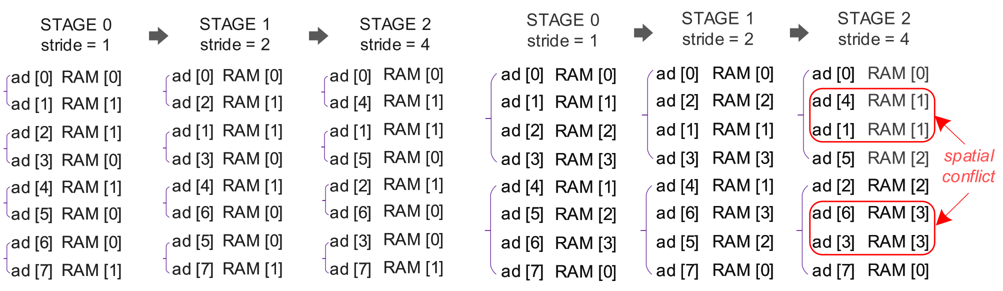


Figure 1: The temporal conflict in pipelined in-place NTT architecture.

two stages is the point at which conflict is most likely to occur. The first-round butterfly operation of the last stage will read $\frac{N}{2}$-stride point pairs [CMV+15], which are computed in the penultimate stage at time slot $\frac{N}{4d}$. To avoid this conflict and achieve 100% usage of NTT processor, the pipeline depth ought to meet the condition: $\frac{N}{4d} \geq \textit{pipeline depth}$. This condition will become more stringent when considering the higher radix NTT. Fortunately, the data vector size in PQC and FHE algorithm is from hundreds to millions magnitude, so this condition can be met under common situations.

Spatial conflict. In every stage of NTT computation, if multiple data points are mapped onto the same bank, we need to perform several read or write operations on a single bank address simultaneously, which results in the so-called spatial conflict. [Joh92] proposes a memory mapping scheme for arbitrary radix in-place FFT hardware, which is described in Figure 2(a) taking 8-point radix-2 FFT as an example. However, [ZYC+20] and [XMX17] point out that this scheme is not suitable for the case when placing multiple butterfly units in every stage. This spatial conflict is shown in Figure 2(b) using 8-point radix-2 FFT with two butterfly units as an example. At the first round of stage 2, four row addresses {0,4,1,5} are mapped onto the bank indexes {0,1,1,2}. It is obvious that raw addresses {4,1}



(a) Radix-2 in-place NTT with d = 1. (b) Radix-2 in-place NTT with d = 2.

Figure 2: The spatial conflict in 8-point radix-2 in-place NTT.

are mapped on to the same bank index 1, which results in the access conflict. Instead of using complex permutation networks as [XHY+20], we will devise a refined and simplified scheme to solve this problem in section 4.

3 Radix-4 DIT NTT and DIF INTT

3.1 DIT Radix-4 NTT with Low Complexity

The pre-processing requires N modular multiplications for each input vector as equation 3 shows. Inspired by the tricks of [ZYC+20] [POG15], we can still avoid this process by merging the $2N$-th roots of unity into every stage of Cooley-Tukey radix-4 DIT NTT algorithm. Thus, the number of modular multiplications is reduced from $(N \log_4 N) + 2N$ to $N \log_4 N$ in the proposed algorithm. Our derivation follows the divide-and-conquer strategy in [Sto06], where the DFT is divided in time domain. Firstly, the pre-processing and the main NTT are written together as belows :

$$A_i = \sum_{j=0}^{N-1} a_j \phi_{2N}^{j} \omega_N^{ij} \bmod q \quad i = 0, 1, ..., N-1 \tag{5}$$

Then, by splitting the summation into four coresidual groups according to the time-domain index j, the equation 6 is transformed as :

$$\begin{aligned} A_i = & \sum_{j=0}^{N/4-1} a_{4j} \phi_{2N}^{4j} \omega_N^{i\cdot(4j)} + \sum_{j=0}^{N/4-1} a_{4j+1} \phi_{2N}^{4j+1} \omega_N^{i\cdot(4j+1)} + \\ & \sum_{j=0}^{N/4-1} a_{4j+2} \phi_{2N}^{4j+2} \omega_N^{i\cdot(4j+2)} + \sum_{j=0}^{N/4-1} a_{4j+3} \phi_{2N}^{4j+3} \omega_N^{i\cdot(4j+3)} \bmod q \\ & i = 0, 1, ..., N-1 \end{aligned} \tag{6}$$

By leveraging the Elimination property of twiddle factor ω and ϕ, the equation 6 can be simplified as :

$$\begin{aligned} A_i = & \sum_{j=0}^{N/4-1} a_{4j} \phi_{N/2}^{j} \omega_{N/4}^{ij} + \omega_N^i \cdot \phi_{2N}^1 \cdot \sum_{j=0}^{N/4-1} a_{4j+1} \phi_{N/2}^{j} \omega_{N/4}^{ij} + \\ & \omega_N^{2i} \cdot \phi_{2N}^2 \cdot \sum_{j=0}^{N/4-1} a_{4j+2} \phi_{N/2}^{j} \omega_{N/4}^{ij} + \omega_N^{3i} \cdot \phi_{2N}^3 \cdot \sum_{j=0}^{N/4-1} a_{4j+3} \phi_{N/2}^{j} \omega_{N/4}^{ij} \bmod q \\ & i = 0, 1, ..., N/4-1 \end{aligned} \tag{7}$$

For simplicity, let F_0, F_1, F_2 and F_3 denote the four summation items $\sum_{j=0}^{N/4-1} a_{4j}\phi_{N/2}^{j}\omega_{N/4}^{ij}$, $\sum_{j=0}^{N/4-1} a_{4j+1}\phi_{N/2}^{j}\omega_{N/4}^{ij}$, $\sum_{j=0}^{N/4-1} a_{4j+2}\phi_{N/2}^{j}\omega_{N/4}^{ij}$, $\sum_{j=0}^{N/4-1} a_{4j+3}\phi_{N/2}^{j}\omega_{N/4}^{ij}$, respectively. Then, the equation 7 is written as :

$$A_i = F_0 + \omega_N^i \cdot \phi_{2N}^1 \cdot F_1 + \omega_N^{2i} \cdot \phi_{2N}^2 \cdot F_2 + \omega_N^{3i} \cdot \phi_{2N}^3 \cdot F_3 \quad i = 0, 1, ..., N/4-1 \tag{8}$$

Obviously, by applying the Periodicity property of twiddle factor, the other three equations can be obtained as:

$$\begin{aligned} &A_{i+N/4} = F_0 + \omega_N^i \cdot \omega_4^1 \cdot \phi_{2N}^1 \cdot F_1 + \omega_N^{2i} \cdot \omega_4^2 \cdot \phi_{2N}^2 \cdot F_2 + \omega_N^{3i} \cdot \omega_4^3 \cdot \phi_{2N}^3 \cdot F_3 \\ &A_{i+2N/4} = F_0 + \omega_N^i \cdot \omega_4^2 \cdot \phi_{2N}^1 \cdot F_1 + \omega_N^{2i} \cdot (\omega_4^2)^2 \cdot \phi_{2N}^2 \cdot F_2 + \omega_N^{3i} \cdot (\omega_4^3)^2 \cdot \phi_{2N}^3 \cdot F_3 \\ &A_{i+3N/4} = F_0 + \omega_N^i \cdot \omega_4^3 \cdot \phi_{2N}^1 \cdot F_1 + \omega_N^{2i} \cdot (\omega_4^2)^3 \cdot \phi_{2N}^2 \cdot F_2 + \omega_N^{3i} \cdot (\omega_4^3)^3 \cdot \phi_{2N}^3 \cdot F_3 \\ &i = 0, 1, ..., N/4-1 \end{aligned} \tag{9}$$

According to the Sum property and Binary property, we can derive that $\omega_4^3 = -\omega_4^1$, $\omega_4^2 = -\omega_4^0 = -1$. So we can express the equation 8 and 9 in the form of matrix as belows:

$$\begin{bmatrix} A_i \\ A_{i+N/4} \\ A_{i+2N/4} \\ A_{i+3N/4} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^1 & -1 & -\omega_4^1 \\ 1 & -1 & 1 & -1 \\ 1 & -\omega_4^1 & -1 & \omega_4^1 \end{bmatrix} \times \left(\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \omega_N^i \cdot \phi_{2N}^1 \\ \omega_N^{2i} \cdot \phi_{2N}^2 \\ \omega_N^{3i} \cdot \phi_{2N}^3 \end{bmatrix} \right) \quad i = 0, 1, ..., N/4-1 \tag{10}$$

Note that F_0, F_1, F_2 and F_3 are actually the four $N/4$-point NTTs, essentially similar to equation 6. By adopting this divide-and-conquer strategy recursively until $\frac{N}{4}$ 4-points, we can obtain the radix-4 NTT algorithm. Furthermore, by merging the powers of N-th roots of unity ω_N and $2N$-th roots of unity ϕ_{2N} in equation 10, we can obtain that:

$$\begin{bmatrix} A_i \\ A_{i+N/4} \\ A_{i+2N/4} \\ A_{i+3N/4} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^1 & -1 & -\omega_4^1 \\ 1 & -1 & 1 & -1 \\ 1 & -\omega_4^1 & -1 & \omega_4^1 \end{bmatrix} \times \left(\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \phi_{2N}^{2i+1} \\ \phi_{2N}^{2(2i+1)} \\ \phi_{2N}^{3(2i+1)} \end{bmatrix} \right) \quad i = 0, 1, ..., N/4-1 \tag{11}$$

Up to now, we just need to precompute the powers of ϕ_{2N} for storage, eliminating the pre-processing of $2N$ modular multiplication completely.

3.2 DIF Radix-4 INTT with Low Complexity

By merging the scale factor N^{-1} and powers of $2N$-th roots of unity ϕ_{2N}^{-i} into every stage, the number of modular multiplications in radix-4 INTT algorithm can be reduced from $N\log_4 N + N$ to $N\log_4 N$ [ZYC+20] [POG15]. At this time, we will adopt the strategy in [GS66], where the DFT is divided in frequency domain and it turns out the so-called Gentleman-Sande Butterfly. At the very beginning, the post-processing and the main INTT are written together as:

$$a_i = N^{-1} \cdot \phi_{2N}^{-i} \cdot \sum_{j=0}^{N-1} A_j \omega_N^{-ij} \bmod q \quad i = 0, 1, ..., N-1 \tag{12}$$

According to the index i, we can split the summation into four consecutive groups with the same size $N/4$. Then, the equation 12 is transformed as:

$$\begin{aligned} a_i = N^{-1} \cdot (\phi_{2N}^{-i} \cdot \sum_{j=0}^{N/4-1} A_j \phi_{2N}^{-j} \omega_N^{-ij} + \phi_{2N}^{-i} \cdot \sum_{j=N/4}^{2N/4-1} A_j \omega_N^{-ij} + \\ \phi_{2N}^{-i} \cdot \sum_{j=2N/4}^{3N/4-1} A_j \omega_N^{-ij} + \phi_{2N}^{-i} \cdot \sum_{j=3N/4}^{N-1} A_j \omega_N^{-ij}) \bmod q \quad i = 0, 1, ..., N-1 \end{aligned} \tag{13}$$

Based on the Periodicity and Binary property of twiddle factor, we can put the four summation items into the same range $[0, N/4-1]$:

$$\begin{aligned} a_i = N^{-1} \cdot (\phi_{2N}^{-i} \cdot \sum_{j=0}^{N/4-1} A_j \omega_N^{-ij} + \phi_{2N}^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+N/4} \omega_N^{-i\cdot(j+N/4)} + \\ \phi_{2N}^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+2N/4} \omega_N^{-i\cdot(j+2N/4)} + \phi_{2N}^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+3N/4} \omega_N^{-i\cdot(j+3N/4)}) \bmod q \\ i = 0, 1, ..., N/4-1 \end{aligned} \tag{14}$$

Because of the Elimination property, the equation 14 can be simplified as:

$$\begin{aligned} a_i = N^{-1} \cdot (\phi_{2N}^{-i} \cdot \sum_{j=0}^{N/4-1} A_j \omega_N^{-ij} + \phi_{2N}^{-i} \cdot \omega_4^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+N/4} \omega_N^{-ij} + \\ \phi_{2N}^{-i} \cdot \omega_4^{-2i} \cdot \sum_{j=0}^{N/4-1} A_{j+2N/4} \omega_N^{-ij} + \phi_{2N}^{-i} \cdot \omega_4^{-3i} \cdot \sum_{j=0}^{N/4-1} A_{j+3N/4} \omega_N^{-ij}) \bmod q \\ i = 0, 1, ..., N/4-1 \end{aligned} \tag{15}$$

Note that the factors ω_4^{-i}, ω_4^{-2i}, ω_4^{-3i} can be classified into four cases according to the coresidual groups of index i:

$$\omega_4^{-i} = \begin{cases} 0 & i = 4r \\ \omega_4^{-1} & i = 4r+1 \\ -1 & i = 4r+2 \\ -\omega_4^{-1} & i = 4r+3 \end{cases} \omega_4^{-2i} = \begin{cases} 1 & i = 4r \\ -1 & i = 4r+1 \\ 1 & i = 4r+2 \\ -1 & i = 4r+3 \end{cases} \omega_4^{-3i} = \begin{cases} 0 & i = 4r \\ \omega_4^{-1} & i = 4r+1 \\ -1 & i = 4r+2 \\ -\omega_4^{-1} & i = 4r+3 \end{cases}$$
$$r = 0, 1, ..., N/4-1 \tag{16}$$

As a result, the equation 16 can be grouped into four parts based on the coresidual groups of index i, correspondingly. With the Elimination property of twiddle factor, the first part can be further written as:

$$\begin{aligned} a_{4i} = (\frac{N}{4})^{-1} \cdot (\frac{1}{4} \cdot \phi_{N/2}^{-i} \cdot \sum_{j=0}^{N/4-1} A_j \omega_{N/4}^{-ij} + \frac{1}{4} \cdot \phi_{N/2}^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+N/4} \omega_{N/4}^{-ij} + \\ \frac{1}{4} \cdot \phi_{N/2}^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+2N/4} \omega_{N/4}^{-ij} + \frac{1}{4} \cdot \phi_{N/2}^{-i} \cdot \sum_{j=0}^{N/4-1} A_{j+3N/4} \omega_{N/4}^{-ij}) \bmod q \\ i = 0, 1, ..., N/4-1 \end{aligned} \tag{17}$$

For simplicity, let G_0,G_1,G_2,G_3 denote the four summation items, respectively.

$$G_m = (\frac{N}{4})^{-1} \cdot \phi_{N/2}^{-i} \cdot \sum_{j=0}^{N/4-1} \frac{1}{4} A_{j+mN/4} \phi_{N/2}^{-i} \omega_{N/4}^{-ij} \quad m = 0, 1, 2, 3 \tag{18}$$

Thus, the four parts of equation 16 can be expressed as:

$$\begin{aligned} & a_{4i} = G_0 + G_1 + G_2 + G_3 \\ & a_{4i+1} = \phi_{2N}^{-1} \cdot \omega_N^{-j} \cdot (G_0 + \omega_4^{-1} G_1 + \omega_4^{-2} G_2 + \omega_4^{-3} G_3) \\ & a_{4i+2} = \phi_{2N}^{-2} \cdot \omega_N^{-2j} \cdot \left[G_0 + \omega_4^{-2} G_1 + (\omega_4^{-2})^2 G_2 + (\omega_4^{-3})^2 G_3\right] \\ & a_{4i+3} = \phi_{2N}^{-3} \cdot \omega_N^{-3j} \cdot \left[G_0 + \omega_4^{-3} G_1 + (\omega_4^{-2})^3 G_2 + (\omega_4^{-3})^3 G_3\right] \\ & i = 0, 1, ..., N/4-1 \end{aligned} \tag{19}$$

With the Sum and Binary property, we can derive that $\omega_4^{-3} = -\omega_4^{-1}$, $\omega_4^{-2} = -\omega_4^0 = -1$. Thus, the equation 20 is further expressed in matrix form:

$$\begin{bmatrix} a_{4i} \\ a_{4i+1} \\ a_{4i+2} \\ a_{4i+3} \end{bmatrix} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^{-1} & -1 & -\omega_4^{-1} \\ 1 & -1 & 1 & -1 \\ 1 & -\omega_4^{-1} & -1 & \omega_4^{-1} \end{bmatrix} \times \begin{bmatrix} G_0 \\ G_1 \\ G_2 \\ G_3 \end{bmatrix} \right) \cdot \begin{bmatrix} 1 \\ \omega_N^{-j} \cdot \phi_{2N}^{-1} \\ \omega_N^{-2j} \cdot \phi_{2N}^{-2} \\ \omega_N^{-3j} \cdot \phi_{2N}^{-3} \end{bmatrix} \tag{20}$$
$$i = 0, 1, ..., N/4 - 1$$

In this way, the N-points INTT can be divided into four $\frac{N}{4}$-points INTTs. By adopting this strategy recursively until $\frac{N}{4}$ 4-points INTTs, we can obtain the DIF radix-4 INTT algorithm. Note that the scale factor is also merged into each stage and the equation 20 can be further written as:

$$\begin{bmatrix} a_{4i} \\ a_{4i+1} \\ a_{4i+2} \\ a_{4i+3} \end{bmatrix} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^{-1} & -1 & -\omega_4^{-1} \\ 1 & -1 & 1 & -1 \\ 1 & -\omega_4^{-1} & -1 & \omega_4^{-1} \end{bmatrix} \times \begin{bmatrix} G_0 \\ G_1 \\ G_2 \\ G_3 \end{bmatrix} \right) \cdot \begin{bmatrix} 1 \\ \phi_{2N}^{-(2j+1)} \\ \phi_{2N}^{-2(2j+1)} \\ \phi_{2N}^{-3(2j+1)} \end{bmatrix} \tag{21}$$
$$i = 0, 1, ..., N/4 - 1$$

By now, we just need to precalculate the powers of $2N$-th roots of unity ϕ_{2N}^{-1} for storage and the post-processing of N-modular multiplication is removed completely.

3.3 Improvement with Divide and Schedule Method

In this section, a versatile schedule method is utilized to reduce the number of modular multiplications and modular additions/subtractions by 20% and 30%, respectively. Thereafter, the complete radix-4 DIT NTT algorithm and radix-4 DIF INTT algorithm with low complexity are presented.

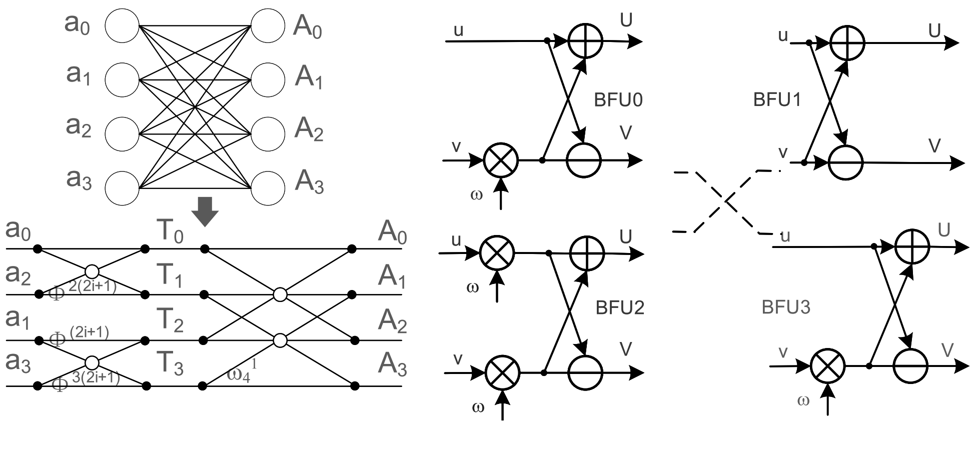


Figure 3: The radix-4 Cookley-Tukey butterfly unit.

Divide and schedule in radix-4 Cooley-Tukey butterfly. When directly calculating the equation 11, it will take 5 modular multiplications, 6 modular additions and 6 modular subtractions. However, based on the divide-and-conquer idea, this computation complexity can be further reduced [HT96]. It is easy to see that some immediate results can be

Algorithm 2 Radix-4 DIT Iterative NTT Algorithm

Input: $\boldsymbol{a}$ denotes a vector of length N, q is modulus and ϕ_{2N} is $2N$-th primitive roots of unity. ω_4^1 is 4-th primitive roots of unity.
Output: $\boldsymbol{A} = DIT\_NTT(\boldsymbol{a})$

```
1: $\boldsymbol{A} \leftarrow$ bit reverse($\boldsymbol{a}$)
2: for $p = 0$ to $\log_4 N - 1$ do
3:     $J \leftarrow 4^p$
4:     $\omega_m \leftarrow \phi_{2N}^{N/(4J)}$
5:     for $k = 0$ to $N/(4J) - 1$ do
6:         for $j = 0$ to $J - 1$ do
7:             $T_0 \leftarrow (A_{4kJ+j} + A_{4kJ+j+2J} \cdot \omega_m^{2(2j+1)}) \bmod q$
8:             $T_1 \leftarrow (A_{4kJ+j} - A_{4kJ+j+2J} \cdot \omega_m^{2(2j+1)}) \bmod q$
9:             $T_2 \leftarrow (A_{4kJ+j+J} \cdot \omega_m^{2j+1} + A_{4kJ+j+3J} \cdot \omega_m^{3(2j+1)}) \bmod q$
10:            $T_3 \leftarrow (A_{4kJ+j+J} \cdot \omega_m^{2j+1} - A_{4kJ+j+3J} \cdot \omega_m^{3(2j+1)}) \bmod q$
11:            $A_{4kJ+j} \leftarrow (T_0 + T_2) \bmod q$
12:            $A_{4kJ+j+J} \leftarrow (T_1 + T_3 \cdot \omega_4^1) \bmod q$
13:            $A_{4kJ+j+2J} \leftarrow (T_0 - T_2) \bmod q$
14:            $A_{4kJ+j+3J} \leftarrow (T_1 - T_3 \cdot \omega_4^1) \bmod q$
15:        end for
16:    end for
17: end for
18: return $\boldsymbol{A}$
```

reused when calculating the radix-4 butterfly operation. Thus, we schedule the matrix multiplication into two-layer operations as belows:

$$\begin{aligned} T_0 &= (F_0 + F_2 \cdot \phi_{2N}^{2(2i+1)}) & A_i &= T_0 + T_2 \\ T_1 &= (F_0 - F_2 \cdot \phi_{2N}^{2(2i+1)}) & A_{i+N/4} &= (T_1 + T_3 \cdot \omega_4^1) \\ T_2 &= (F_1 \cdot \phi_{2N}^{2i+1} + F_3 \cdot \phi_{2N}^{3(2i+1)}) & A_{i+2N/4} &= (T_0 - T_2) \\ T_3 &= (F_1 \cdot \phi_{2N}^{2i+1} - F_3 \cdot \phi_{2N}^{3(2i+1)}) & A_{i+3N/4} &= (T_1 - T_3 \cdot \omega_4^1) \end{aligned} \tag{22}$$

At this time, the radix-4 butterfly operation is reduced to 4 modular multiplications, modular additions and modular subtractions, which results in the two-layer Cooley-Tukey butterfly unit. Figure 3 depicts this change intuitively and presents the detailed architecture. As a result, the complete two-layer radix-4 DIT NTT with low complexity is demonstrated in algorithm 2.

Divide and schedule in radix-4 Gentleman-Sande butterfly. Similar to radix-4 NTT, the complexity in radix-4 INTT butterfly operation can also be further reduced. Using straightforward method to compute the equation 21 still requires 5 modular multiplications, 6 modular additions and 6 modular subtractions. Thus, by scheduling the matrix multiplication to reuse the immediate result, we obtain that:

$$\begin{aligned} T_0 &= F_0 + F_2 & a_{4i} &= T_0 + T_2 \\ T_1 &= F_0 - F_2 & a_{4i+1} &= (T_1 + T_3) \cdot \phi_{2N}^{-(2j+1)} \\ T_2 &= F_1 + F_3 & a_{4i+2} &= (T_0 - T_2) \cdot \phi_{2N}^{-2(2j+1)} \\ T_3 &= (F_1 - F_3) \cdot \omega_4^{-1} & a_{4i+3} &= (T_1 - T_3) \cdot \phi_{2N}^{-3(2j+1)} \end{aligned} \tag{23}$$

In this way, we just need 4 modular multiplications, modular additions and modular subtractions, which results in the two-layer Gentleman-Sande butterfly unit. Figure 4 presents the changes and detailed architecture as well. Since the ω_4^1 and ω_4^{-1} are constants, the multiplication in BFU3 and IBFU2 can be alternatively implemented with a series of

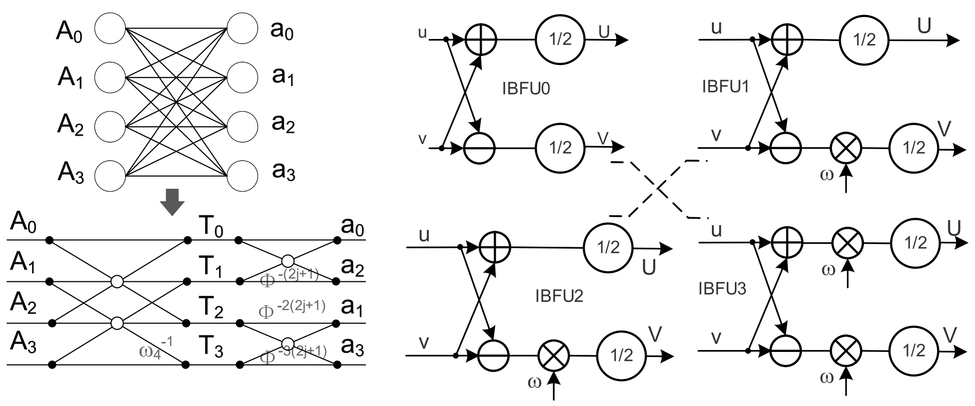


Figure 4: The radix-4 Gentleman-Sande butterfly unit.

modular additions and shifts. So far, the complete two-layer radix-4 DIF INTT with low complexity is demonstrated in algorithm 3.

Algorithm 3 Radix-4 DIF Iterative INTT Algorithm

Input: $\boldsymbol{a}$ denotes a vector of length N, q is modulus and ϕ_{2N}^{-1} is inverse element of $2N$-th primitive roots of unity. ω_4^{-1} is inverse element of 4-th primitive roots of unity.

Output: $\boldsymbol{A} = DIF\_INTT(\boldsymbol{a})$

```
1:  for $p = \log_4 N - 1$ to 0 do
2:      $J \leftarrow 4^p$
3:      $\omega_m \leftarrow \phi_{2N}^{-N/(4J)}$
4:      for $k = 0$ to $N/(4J) - 1$ do
5:          for $j = 0$ to $J - 1$ do
6:              $T_0 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j} + a_{4kJ+j+2J}) \bmod q$
7:              $T_1 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j} - a_{4kJ+j+2J}) \bmod q$
8:              $T_2 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j+J} + a_{4kJ+j+3J}) \bmod q$
9:              $T_3 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j+J} - a_{4kJ+j+3J}) \cdot \omega_4^{-1} \bmod q$
10:             $A_{4kJ+j} \leftarrow \frac{1}{2} \cdot (T_0 + T_2) \bmod q$
11:             $A_{4kJ+j+J} \leftarrow \frac{1}{2} \cdot (T_1 + T_3) \cdot \omega_m^{-(2j+1)} \bmod q$
12:             $A_{4kJ+j+2J} \leftarrow \frac{1}{2} \cdot (T_0 - T_2) \cdot \omega_m^{-2(2j+1)} \bmod q$
13:             $A_{4kJ+j+3J} \leftarrow \frac{1}{2} \cdot (T_1 - T_3) \cdot \omega_m^{-3(2j+1)} \bmod q$
14:         end for
15:     end for
16: end for
17: $\boldsymbol{A} \leftarrow$ bit reverse($\boldsymbol{A}$)
18: return $\boldsymbol{A}$
```

3.4 Avoiding Bit-reversed Cost and Reducing Memory Footprint

When straightly performing the classic in-place DIT NTT and DIF INTT as [ZYC+20] [XHY+20] [MKÖ+20b], bit-reversed operation is required at the very beginning of NTT and ending of INTT, respectively. Because the classic DIT NTT receives the input vector in bit-reversed order and produces the output vector in natural order. The classic DIF INTT is contrary to the orientation of DIT NTT. [POG15] [XL21] [CG99] avoid the bit-reversed operation for radix-2 DIT NTT and DIF INTT by rearranging the loop structure and storage structure of twiddle factors. In this section, we will generalize this bit-reversed issue for arbitrary radix DIT NTT and DIF INTT and propose the bit-reversed-free radix-4

DIT NTT and DIF INTT algorithm, respectively. We use the notation NR (RN) as belows to represent that the input vector is received in natural (bit-reversed) order and the output vector is produced in bit-reversed (natural) order. In addition, we will show how the memory footprint can be reduced by reusing the twiddle factors of forward radix-2/4 NTT in the inverse radix-2/4 NTT.

3.4.1 Proposed DIT-NR Radix-4 NTT

In DIT NTT, the bit-reversed-free trait indicates that the input vector is received in natural order and the output vector is produced in bit-reversed order, which is opposite to the classic situation. [CG99] obtains the radix-2 bit-reversed-free DIT NTT by changing the loop structure and the storage method of twiddle factors. Inspired by this work, we derive the radix-4 bit-reversed-free DIT NTT algorithm. Based on this derivation, the bit-reversed operation in arbitrary radix NTT can be removed. This optimization technique is achieved by three important observations as belows:

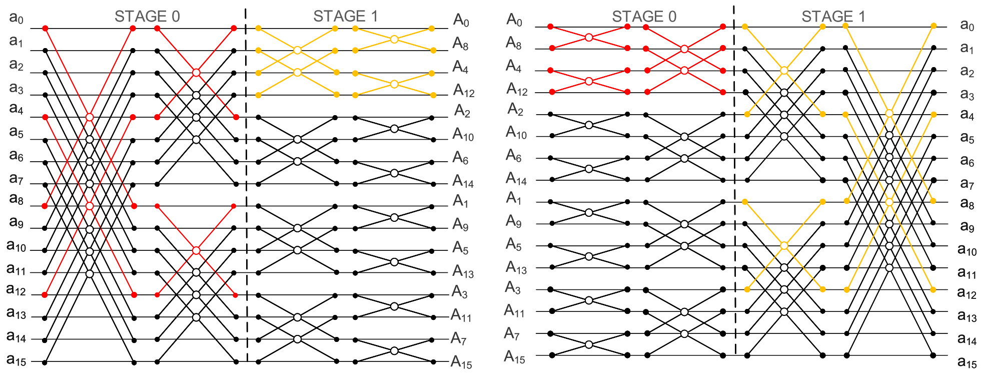
(1) In analogy with classic DIF INTT algorithm, the address with natural order in DIT NTT can be generated by just reversing the first loop.

(2) To still obtain the correct results after reversing the first loop, the generation of twiddle factors is required to rearranged accordingly.

(3) After performing the operation in (1), the iterative principle of index j and k in algorithm 2 will exactly exchange with each other.

Based on these insights, the arrangement of twiddle factors in item (2) can be realized with three steps based on item (3). The first step is to move the place of generating twiddle factors from the innermost loop j to the middle loop k. The second step is to replace the index j in exponential position of twiddle factors in algorithm 2 with bit-reversed index k. The final step is to replace the powers $N/(4J)$ in algorithm 2 line 4 with powers J. It is worth to mention that this method can also be applied in DIT NTT of other radices to avoid the bit-reversed cost. The detailed changes in terms of the generation of twiddle factors are described in equation 24.

$$\begin{aligned} \phi_{2N}^{N/(4J)} &\Rightarrow \phi_{2N}^{J} & \omega_m^{2j+1} &\Rightarrow \omega_m^{2\text{bit-reversed}(k)+1} \\ \omega_m^{2(2j+1)} &\Rightarrow \omega_m^{2\cdot(2\text{bit-reversed}(k)+1)} & \omega_m^{3(2j+1)} &\Rightarrow \omega_m^{3\cdot(2\text{bit-reversed}(k)+1)} \end{aligned} \tag{24}$$



(a) The data flow of radix-4 DIT-NR NTT. (b) The data flow of radix-4 DIF-RN INTT.

Figure 5: The data flow of 16 points radix-4 NTT/INTT.

In the hardware implementation, the twiddle factors are actually precomputed and stored in the ROM. Hence, we propose the complete bit-reversed-free radix-4 DIT NTT in algorithm 4 along with the method of precomputing twiddle factors. Figure 5(a) presents

Algorithm 4 Radix-4 DIT-NR NTT Algorithm

Input: $\boldsymbol{a}$ denotes a vector of length N, q is modulus and ϕ_{2N} is $2N$-th primitive roots of unity. ω_4^1 is 4-th primitive roots of unity.
Output: $\boldsymbol{A}$ = DIT-NR NTT($\boldsymbol{a}$)

```
1: **Precompute the twiddle factors:**
2: for $p = \log_4 N - 1$ to 0 do
3:     $J \leftarrow 4^p$
4:     $\omega_m \leftarrow \phi_{2N}^J$
5:     for $k = 0$ to $N/(4J) - 1$ do
6:         $\omega a1$_ROM.append $[\omega_m^{2\text{bit-reversed}(k)+1}]$
7:         $\omega a2$_ROM.append $[\omega_m^{2\cdot(2\text{bit-reversed}(k)+1)}]$
8:         $\omega a3$_ROM.append $[\omega_m^{3\cdot(2\text{bit-reversed}(k)+1)}]$
9:     end for
10: end for
11: **Perform the radix-4 DIT-NR NTT:**
12: for $p = \log_4 N - 1$ to 0 do
13:     $J \leftarrow 4^p$
14:     $r \leftarrow 0$
15:     for $k = 0$ to $N/(4J) - 1$ do
16:         $\omega_1 \leftarrow \omega a1$_ROM $[r]$
17:         $\omega_2 \leftarrow \omega a2$_ROM $[r]$
18:         $\omega_3 \leftarrow \omega a3$_ROM $[r]$
19:         $r \leftarrow r + 1$
20:         for $j = 0$ to $J - 1$ do
21:             $T_0 \leftarrow (a_{4kJ+j} + a_{4kJ+j+2J} \cdot \omega_2) \bmod q$
22:             $T_1 \leftarrow (a_{4kJ+j} - a_{4kJ+j+2J} \cdot \omega_2) \bmod q$
23:             $T_2 \leftarrow (a_{4kJ+j+J} \cdot \omega_1 + a_{4kJ+j+3J} \cdot \omega_3) \bmod q$
24:             $T_3 \leftarrow (a_{4kJ+j+J} \cdot \omega_1 - a_{4kJ+j+3J} \cdot \omega_3) \bmod q$
25:             $A_{4kJ+j} \leftarrow (T_0 + T_2) \bmod q$
26:             $A_{4kJ+j+J} \leftarrow (T_1 + T_3 \cdot \omega_4^1) \bmod q$
27:             $A_{4kJ+j+2J} \leftarrow (T_0 - T_2) \bmod q$
28:             $A_{4kJ+j+3J} \leftarrow (T_1 - T_3 \cdot \omega_4^1) \bmod q$
29:         end for
30:     end for
31: end for
32: return $\boldsymbol{A}$
```

the corresponding 16-points dataflow of radix-4 DIT-NR NTT where the input vector is received in natural order and the output vector is produced in bit-reversed order. The lines denoted with red and yellow color represent the first-round butterfly operation in stage 0 and stage 1, respectively.

3.4.2 Proposed DIF-RN Radix-4 INTT

In DIF-RN INTT, the bit-reversed-free trait indicates that the input vector is received in bit-reversed order and the output vector is produced in natural order, which is also opposite to the classic DIF INTT. Based on DIT-NR NTT, we can obtain the DIF-RN INTT with a direct method by just reversing the first loop in algorithm 3 line 1 and replacing the twiddle factor in equation 24 with its inverse element. Nevertheless, this method cannot lead to a perfect implementation in terms of memory footprint. As shown in the reference software implementation of Kyber from NIST, the twiddle factors in radix-2 DIF INTT can reuse the twiddle factors of radix-2 DIT INTT by leveraging the Binary property: $\phi_{2N}^N = -1 \bmod q \Rightarrow \phi_{2N}^{-i} = -\phi_{2N}^N \cdot \phi_{2N}^{-i} \bmod q = -\phi_{2N}^{N-i} \bmod q$. In this section, we show that the twiddle factors in radix-4 DIT-NR NTT can also be reused in radix-4 DIF-RN INTT by making full use of the properties of twiddle factors and suitable schedule. Firstly, to transform the negative exponent of ϕ_{2N} into positive field, we derive three tricks shown in equation 25 from the Binary and Elimination property of twiddle factors:

$$\begin{aligned}
&\phi_{2N}^{-i} \bmod q = \phi_{2N}^{-N/2} \cdot \phi_{2N}^{N/2-i} \bmod q = \omega_4^{-1} \cdot \phi_{2N}^{N/2-i} \bmod q = -\omega_4^{1} \cdot \phi_{2N}^{N/2-i} \bmod q \\
&\phi_{2N}^{-i} \bmod q = -\phi_{2N}^{N} \cdot \phi_{2N}^{N-i} \bmod q = -\phi_{2N}^{N-i} \bmod q \\
&\phi_{2N}^{-i} \bmod q = \phi_{2N}^{-3N/2} \cdot \phi_{2N}^{3N/2-i} \bmod q = -\phi_{2N}^{-N/2} \cdot \phi_{2N}^{3N/2-i} \bmod q \\
&\qquad\qquad = -\omega_4^{-1} \cdot \phi_{2N}^{3N/2-i} \bmod q = \omega_4^{1} \cdot \phi_{2N}^{3N/2-i} \bmod q
\end{aligned} \tag{25}$$

Then, we apply these three tricks into equation 24 to obtain the inverse twiddle factor in algorithm 3 as belows:

$$\begin{aligned}
\omega_m^{-(2j+1)} &\Rightarrow -\omega_4^{1} \cdot \omega_m^{N/2-[2\text{bit-reversed}(k)+1]} \\
\omega_m^{-2(2j+1)} &\Rightarrow -\omega_m^{N-[2\cdot(2\text{bit-reversed}(k)+1)]} \\
\omega_m^{-3(2j+1)} &\Rightarrow \omega_4^{1} \cdot \omega_m^{3N/2-[3\cdot(2\text{bit-reversed}(k)+1)]}
\end{aligned} \tag{26}$$

Note that the factor ω_4^1 can be scheduled to multiply with ω_4^{-1} in algorithm 3 line 9. Hence applying the equation 26 into the two-layer butterfly operation of algorithm 3 will not increase the original complexity. To this end, we can reuse the twiddle factors generated in algorithm 4, which turns out the complete bit-reversed-free DIF INTT algorithm 5. As a result, the bit-reversed-free radix-4 Gentleman Sande butterfly unit can be obtained by just exchanging the position of IBFU0 with IBFU2 and swapping the two operands of modular subtracter in Figure 4. Figure 5(b) presents the corresponding 16-points dataflow of radix-4 DIF-RN INTT where the input vector is received in bit-reversed order and the output vector is produced in natural order. The lines denoted with red and yellow color also represent the first-round butterfly operation in stage 0 and stage 1, respectively.

Algorithm 5 Radix-4 DIF-RN INTT Algorithm

Input: $\boldsymbol{a}$ denotes a vector of length N, q is modulus and $\omega a1$_ROM, $\omega a2$_ROM, $\omega a3$_ROM are three arrays generated by algorithm 4.
Output: $\boldsymbol{A}$ = DIF-RN INTT($\boldsymbol{a}$)

```
1: for $p = 0$ to $\log_4 N - 1$ do
2:     $J \leftarrow 4^p$
3:     $r \leftarrow \frac{N-1}{3} - 1$
4:     for $k = 0$ to $N/(4J) - 1$ do
5:         $\omega_1 \leftarrow \omega a1\_ROM\ [r]$
6:         $\omega_2 \leftarrow \omega a2\_ROM\ [r]$
7:         $\omega_3 \leftarrow \omega a3\_ROM\ [r]$
8:         $r \leftarrow r - 1$
9:         for $j = 0$ to $J - 1$ do
10:            $T_0 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j} + a_{4kJ+j+2J}) \bmod q$
11:            $T_1 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j+2J} - a_{4kJ+j}) \cdot \omega_4^1 \bmod q$
12:            $T_2 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j+J} + a_{4kJ+j+3J}) \bmod q$
13:            $T_3 \leftarrow \frac{1}{2} \cdot (a_{4kJ+j+3J} - a_{4kJ+j+J}) \bmod q$
14:            $A_{4kJ+j} \leftarrow \frac{1}{2} \cdot (T_0 + T_2) \bmod q$
15:            $A_{4kJ+j+J} \leftarrow \frac{1}{2} \cdot (T_1 + T_3) \cdot \omega_1 \bmod q$
16:            $A_{4kJ+j+2J} \leftarrow \frac{1}{2} \cdot (T_2 - T_0) \cdot \omega_2 \bmod q$
17:            $A_{4kJ+j+3J} \leftarrow \frac{1}{2} \cdot (T_3 - T_1) \cdot \omega_3 \bmod q$
18:        end for
19:    end for
20: end for
21: return $\boldsymbol{A}$
```

4 Conflict-free Memory Mapping Scheme

By applying loop unroll technique, a scalable iterative NTT algorithm is proposed in section 4.1. Thereafter, the general and efficient conflict-free memory mapping scheme is presented in section 4.2. Based on the scalable algorithm and memory mapping scheme, the NTT multiplication architecture can be parallelized.

Algorithm 6 Scalable Iterative NTT Algorithm

```
Input: $\boldsymbol{a}$, $N$, $R$. Here $\boldsymbol{a}$ denotes a vector of length $N$. $d$ is the number of butterfly units. $R$ denotes the radix of NTT.
Output: $\boldsymbol{A} = Scalable\_NTT(\boldsymbol{a})$
1: for $p = \log_R N - 1$ to 0 do
2:     $J \leftarrow R^p$
3:     if $J < d$ then
4:         for $k = 0$ to $N/(Rd) - 1$ do
5:             for $i = 0$ to $d/J - 1$ do
6:                 for $j = 0$ to $J - 1$ do
7:                     $\begin{bmatrix} A_0 \\ A_1 \\ ... \\ A_{R-1} \end{bmatrix} = BF\left(\begin{bmatrix} a_{kRd+iRJ+j} \\ a_{kRd+iRJ+j+J} \\ ... \\ a_{kRd+iRJ+j+(R-1)J} \end{bmatrix}\right)$
8:                 end for
9:             end for
10:        end for
11:    else
12:        for $k = 0$ to $N/(RJ) - 1$ do
13:            for $i = 0$ to $J/d - 1$ do
14:                for $j = 0$ to $d - 1$ do
15:                    $\begin{bmatrix} A_0 \\ A_1 \\ ... \\ A_{R-1} \end{bmatrix} = BF\left(\begin{bmatrix} a_{kRJ+id+j} \\ a_{kRJ+id+j+J} \\ ... \\ a_{kRJ+id+j+(R-1)J} \end{bmatrix}\right)$
16:                end for
17:            end for
18:        end for
19:    end if
20: end for
21: return $\boldsymbol{A}$
```

4.1 Scalable Iterative NTT Algorithm

Determining the order. At the very beginning, we determine the way how the data points are managed in parallel computation. Taking DIT-NR radix-2 NTT with 2 butterfly units as an example, the 8 points data flow along with index set $\{p, k, j\}$ is depicted in Figure 6. Let $G^0_{p,k,j}$ and $G^1_{p,k,j}$ denote the upper and lower point in stage p, group k and round j, respectively. The butterfly operations denoted with the same color belong to the same group. Then, we determine that the parallel 4 data points in the second stage of 8-points INTT are fetched and stored in the following order: $\{G^0_{1,0,0}, G^1_{1,0,0}, G^0_{1,0,1}, G^1_{1,0,1}\} \rightarrow \{G^0_{1,1,0}, G^1_{1,1,0}, G^0_{1,1,1}, G^1_{1,1,1}\}$. It is easy to see that the parallel data points are managed gradually according to the growing order of index k. For 16 points radix-4 NTT with 8 parallel butterfly units, one can also utilize the similar method with radix-2 to fetch and store the 8 data points.

Scalable NTT algorithm. We derive and present the scalable iterative algorithm for arbitrary radix-R NTT based on the determined address order. However, applying the loop unroll method in iterative NTT algorithm directly can not lead to a perfect implementation. Because the range of index j, k in the inner loop and middle loop depends on the index p in the outer loop. The relative size of d and J will influence the number of iterations. As a result, the loop unroll method is classified into 2 cases: (1) $J < d$, the parallel data sets

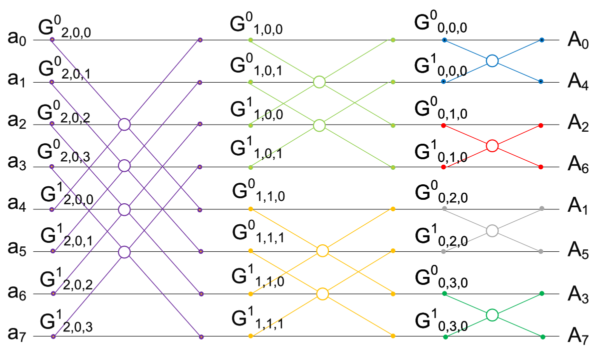


Figure 6: The data flow of 8 points radix-2 DIT-NR NTT.

Algorithm 7 Conflict-free memory mapping scheme for arbitrary radix in-place NTT.

Input: a, N, r^k, d. Here a denotes the old address to be mapped. N denotes the total number of data points. r is a prime and $r^k = R$ denotes the radix of NTT. d represents the number of parallel butterfly units and it also stands for step size.

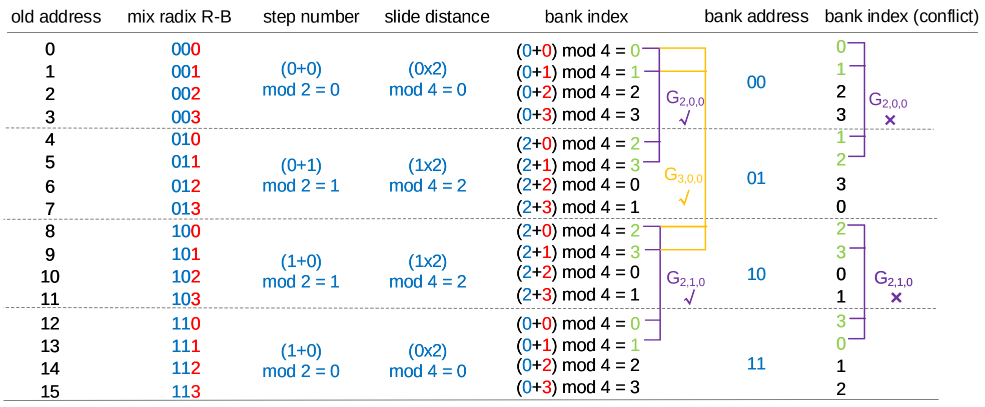
Output: BI, BA. Here BI denotes the bank index into which the old address is mapped. BA denotes the offset of the storage location within each inner bank.

```
1: Let $B$ denote the number of banks, where $B$ satisfies the condition $B = R \times d$ and $N \mid B$ in the interleaved memory system.
2: Calculating the total digit width $T$ of old address expressed in radix r: $T = log_r N$.
3: Calculating the digit width of bank number value expressed in radix r: $M = log_r B = log_r(R \times d)$.
4: Computing the digit width of step size expressed in radix r: $C = T - M$.
5: Then the old address $a$ can be expressed in radix r as:
6:         $a = [a_{T-1}, a_{T-2}, ..., a_1, a_0]_r = [a_{T-1}, ..., a_{T-C}, a_{M-1}, ..., a_1, a_0]_r$.
7: Expressing the lower part $[a_{M-1}, ..., a_1, a_0]_r$ in radix $B$ as $[b]_B$.
8: Expressing the higher part $[a_{T-1}, ..., a_{T-C}]_r$ in radix $R = r^k$ as $[b_{m-1}, b_{m-2}, ..., b_0]_R$, where $m$ is the digit width after radix R transformation.
9: Then the old address can be expressed in mixed radix $B$-$R$ as $a = [b_{m-1}, b_{m-2}, ..., b_0]_R \, [b]_B$.
10: Defining the step number as $SN = (b_{m-1} + b_{m-2} + ... + b_0) \bmod R$.
11: Then, the bank index for old address is calculated as: $BI = (b + SN \cdot d) \bmod B$.
12: The new bank address for old address is calculated as: $BA = a >> M = [a_{T-1}, ..., a_{T-C}]_r$.
13: return $BI, BA$.
```

will contain several iterations of inner loop. (2) $J \geq d$, the inner loop will cover several parallel data sets. The scalable iterative NTT algorithm is illustrated in algorithm 6. For brevity, the concrete butterfly operation within the innermost loop is left out while the address generation method is kept.

4.2 Proposed Solution to Conflict Issue

To address the spatial conflict in section 2.2.2, we propose a new and efficient memory mapping scheme as algorithm 7 shows. The mixed radix expression divides the old address into the higher part and the lower part, respectively. It is easy to see that this memory mapping method can be implemented by a few XOR and shift operations if the radix is power-of-2. In addition, it can support arbitrary radix in-place NTT algorithm, conducing to the non-radix-2 NTT hardware design. Taking 16 points radix-2 NTT with 2 parallel butterfly units as an example, the detailed mapping process is described in Figure 7. According to the proposed memory mapping scheme, the 16 old addresses with 4 bit width is expressed as mixed radix 2-4 form as the second column shows. Then the so-called step number is obtained by accumulating the digit in the higher part of old address following a modular reduction by 2 ($R = 2$) as shown in the third column. The value of slide distance depends on the step number (SN) and the step size 2 ($d = 2$), which is calculated as $(SN \times d)$ mod B. Finally, the bank index is handily obtained by adding the lower part to the slide distance following a modular reduction by 4 ($B = 4$). As a result, at stage 0 ($p = 2$), the 4 parallel data points in group 0 round 0 ($G_{2,0,0}$) and group 1 round 0 ($G_{2,1,0}$) are mapped onto 4 different banks $\{0, 1, 2, 3\}$ and $\{2, 3, 0, 1\}$, respectively. By following the memory mapping scheme ahead, the validity in other radices can be confirmed as well. To make contrast with the memory mapping scheme in [Joh92] [ZYC+20], the final column presents the corresponding bank mapping results, which leads to the memory conflict mentioned in section 2.2.2. To highlight the wider applicability of this conflict-free memory mapping scheme, Figure 15 in appendix also depicts the detailed mapping process for 27 points radix-3 in-place NTT.



old address	mix radix R-B	step number	slide distance	bank index	bank address	bank index (conflict)
0	000	(0+0) mod 2 = 0	(0x2) mod 4 = 0	(0+0) mod 4 = 0	00	0
1	001			(0+1) mod 4 = 1		1
2	002			(0+2) mod 4 = 2		2
3	003			(0+3) mod 4 = 3		3
4	010	(0+1) mod 2 = 1	(1x2) mod 4 = 2	(2+0) mod 4 = 2	01	1
5	011			(2+1) mod 4 = 3		2
6	012			(2+2) mod 4 = 0		3
7	013			(2+3) mod 4 = 1		0
8	100	(1+0) mod 2 = 1	(1x2) mod 4 = 2	(2+0) mod 4 = 2	10	2
9	101			(2+1) mod 4 = 3		3
10	102			(2+2) mod 4 = 0		0
11	103			(2+3) mod 4 = 1		1
12	110	(1+0) mod 2 = 0	(0x2) mod 4 = 0	(0+0) mod 4 = 0	11	3
13	111			(0+1) mod 4 = 1		0
14	112			(0+2) mod 4 = 2		1
15	113			(0+3) mod 4 = 3		2

Figure 7: The detailed memory mapping scheme for 16 points radix-2 NTT with d = 2.

5 Scalable Radix-2/4 NTT Multiplication Architecture

5.1 The Overall Scalable Architecture

In this section, we propose the scalable radix-2 and radix-4 NTT multiplication architecture which can be instantiated for different vector size N, modulus q and number of parallel butterfly units d. As shown in Figure 8, the overall architecture is composed of eight different modules. The address generator is mainly implemented by the counter and shift logic gates. The memory mapping unit consists of a few XOR logic gates and 4-to-1 or 2-to-1 multiplexers in radix-2 and radix-4 NTT, respectively. The arbiter module decodes the bank indexes as corresponding selection signals to control the path of three interconnect networks. The number of fan-ins in radix-2 NTT interconnect network is fourfold as large as that of the radix-4 NTT. Because the number of banks in radix-2 NTT will double compared to radix-4 NTT under the same configuration of butterfly numbers. This difference will influence the performance between radix-2 and radix-4 NTT. In section 6.1, we will quantify this difference in detail. Another distinguishing feature is that the one-dimension butterfly unit in radix-2 NTT is extended in two-dimension array in radix-4 NTT, which elaborates the aforementioned difference in terms of bank number. The twiddle factors of NTT are precomputed and stored in a ROM in this architecture. Since we reuse the twiddle factors in both radix-2 and radix-4 NTT to obtain the radix-2 and radix-4 INTT, the total types of twiddle factors needs to be stored is just $N-1$ words, which is almost half amount of the state-of-the-art designs like [XHY+20] and [ZYC+20]. The proposed radix-4 NTT core requires vector length N to be powers-of-4, which can be applied in third-round-candidate PQC algorithms like Falcon-1024 (N = 1024, q = 12289), Dilithium (N = 256, $q = 2^{23} - 2^{13} + 1$) and Saber (N = 256, q with prime lift trick) [FBR+21]. While Falcon-512 (N = 512, q = 12289) can resort to the radix-2 NTT core. One can generate the case-oriented NTT hardware architecture with different parallel degree based on the proposed design paradigm. In this context, we take the parameters N = 1024 and q = 14-bit modulus as a case study to implement the radix-2 and radix-4 NTT kernels with different number of butterfly units.

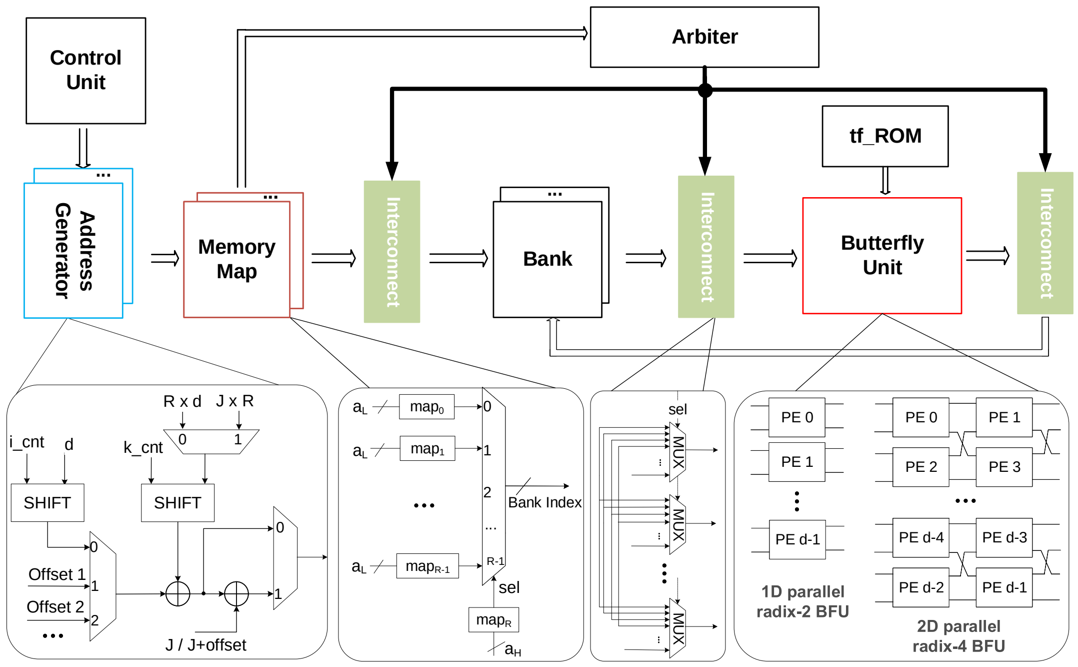


Figure 8: The scalable radix-2/4 NTT multiplication architecture.

5.2 Compact Radix-4 Butterfly Unit

As illustrated in section 3.3, the radix-4 butterfly unit is divided and scheduled into two-layer architecture to further reduce the computation complexity. In this section, we excavate the symmetric operator in radix-4 Cooley-Tukey and Gentleman-Sande butterfly units to reuse the hardware resource. It is observed that the IBFU1 needs an extra modular multiplication by 1/2 and the execution sequence is exactly on the contrary with BFU0. The butterfly unit pairs BFU1 and IBFU0, BFU2 and IBFU3, BFU3 and IBFU2 also have the similar features. As a result, the DIT radix-4 NTT and DIF radix-4 INTT can be performed with 4 configurable processing elements. Figure 9 depicts the specific architecture from PE0 to PE3. When the multiplexer selection signal *sel* is set to 0 or 1, the four processing elements are configured to perform DIT NTT or DIF INTT operation, respectively. So far, we just need 4 modular adders, modular subtracters and modular multipliers to implement the NTT/INTT butterfly operation. For parametric hardware design, Montgomery reduction and Barrett reduction are two commonly used methods. [BGV93] indicates that when the bit width of modulus is less than 32 bits, using Barrett

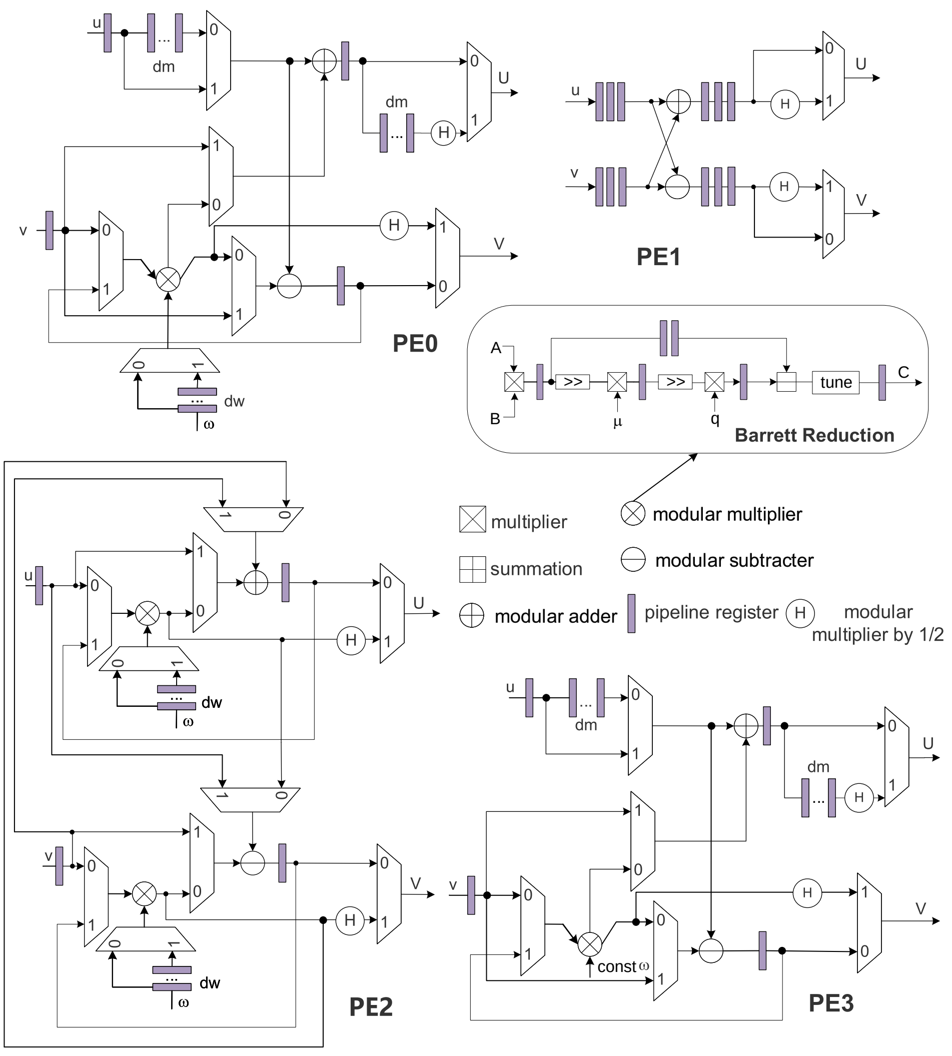


Figure 9: The compact architecture of radix-4 butterfly unit.

reduction is feasible. Thus, our modular multiplier follows Barrett reduction method as shown in Figure 9. The modular multiplier is designed with four pipeline stages and the main critical path is the n-bit multiplication. The parameter μ is defined as $\lfloor 2^{2n}/q \rfloor$ and can be precomputed for configuration. By adding slightly extra control logic, the radix-4 butterfly unit can also perform point-wise multiplication, modular addition/subtraction, which are not depicted for brevity.

Figure 10 shows the configurable routing structure in the processing element array. When the signal *sel_p* is set to 0, the PEs in the first column will receive the data points fetched from the banks and conduct the first-layer butterfly operation. The PEs in the second column will receive the computation results from the first-layer and perform the second-layer butterfly operation. When the signal *sel_p* is set to 1, the configuration is exactly the opposite. By applying this unified PE array, the hardware resource can be reduced by approximately 50% compared to the naive implementation. It is worth to mention that unifying the radix-8 or higher-radix NTT and INTT butterfly operation will be more difficult because the symmetric property of these operators is hard to excavate and the routing path will become more complicated.

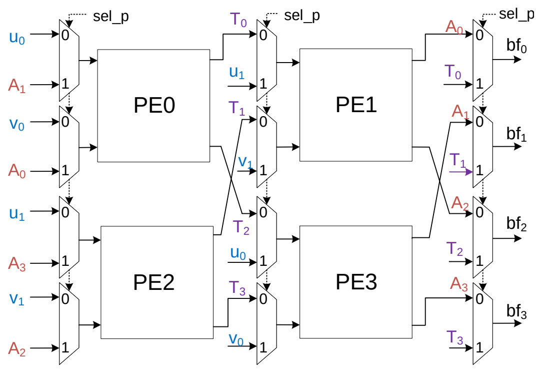


Figure 10: The routing structure of radix-4 PE array.

6 Implementation Results and Comparisons

The proposed architecture is designed with Verilog HDL and implemented on the 28 nm Xilinx Virtex-7 FPGA (xc7vx690tffg1761-3). The resource overhead and highest frequency are obtained from Vivado 2020.2 under the default strategy for synthesis and implementation. In section 6.1, we will make a performance evaluation between radix-4 and radix-2 NTT multiplication architecture through theoretical analysis and implementation results. In section 6.2, we will compare our implementation results of memory mapping scheme and NTT core with the state-of-the-art works.

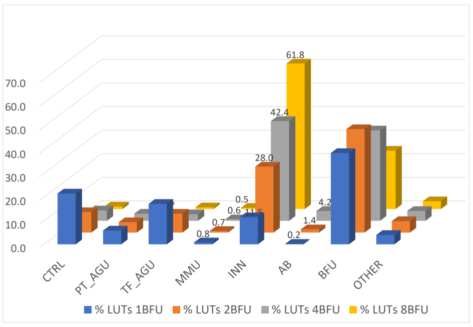
Table 1: The theoretical comparison between radix-2 and radix-4 NTT/INTT. AGU: address generator. MMU: memory mapping unit. INN: interconnect network. AB: Arbiter. BFU: butterfly unit. TF ROM: twiddle factor ROM. MA: modular adder. MS: modular subtracter. MM: modular multiplier. The variation between radix-2 and radix-4 in terms of MUXs is related to the total number of fan-ins.

Type	**AGU**	**MMU**	**AB**	**INN**	**Bank**	**BFU**	**TF ROM**	**NTT/INTT cycle**
radix-2	2n	2n× 2-to-1 MUXs	2n× 2n-to-1 MUXs	2n× 2n-to-1 MUXs	2d	n× MA/MS/MM	$N-1$	$(N/2d)log_2N$
radix-4	n	n× 4-to-1 MUXs	n× n-to-1 MUXs	n× n-to-1 MUXs	n	n× MA/MS/MM*	$N-1$	$(N/d)log_4N$
variation	↓ 50%	–	↓ 75%	↓ 75%	↓ 50%	–	–	–

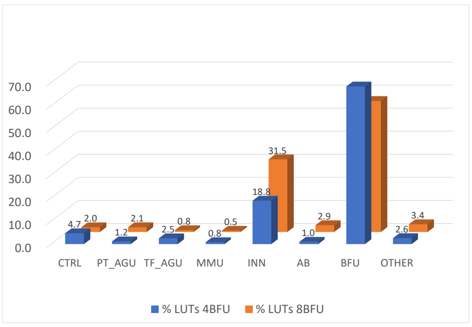
* : Multiplying ω_4^1 (ω_4^{-1}) can be alternatively replaced with a series of modular additions and shifts.
n : number of processing elements.

6.1 Performance Evaluation Between Radix-4 and Radix-2 NTT

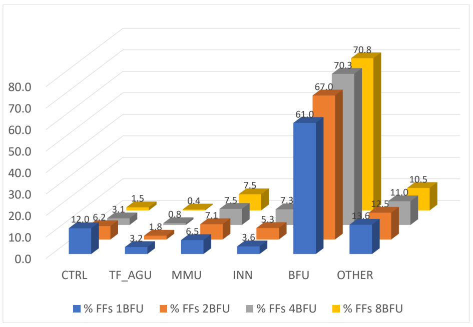
To justify the advantage of radix-4 NTT, we compare its area-time performance with radix-2 NTT through theoretical analysis and implementation results. Table 3 shows the theoretical difference between radix-2 and radix-4 NTT in terms of hardware resources about some common modules. It is observed that the radix-4 NTT consumes the same number of cycles with radix-2 NTT theoretically, but it only needs half number of banks when configured with the same number of butterfly units. The reason for this difference is that the radix-4 NTT performs two consecutive layers of butterfly operations, which halves the concurrent data points to be fetched at every stage. This good balance between intra-stage and inner-stage parallelization heavily reduces the number of fan-ins of interconnect network. As shown in Table 3, the number of fan-ins in the interconnect network of radix-4 NTT is reduced by 75% compared to radix-2 NTT. In addition, the number of fan-outs in the memory mapping unit and address generator unit of radix-4 NTT are both reduced by 75% and 50%, respectively.



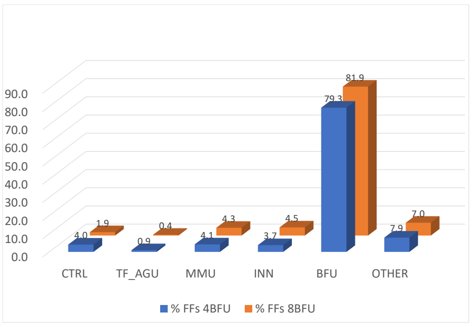
(a) LUTs proportion in radix-2 NTT



(b) LUTs proportion in radix-4 NTT



(c) FFs proportion in radix-2 NTT

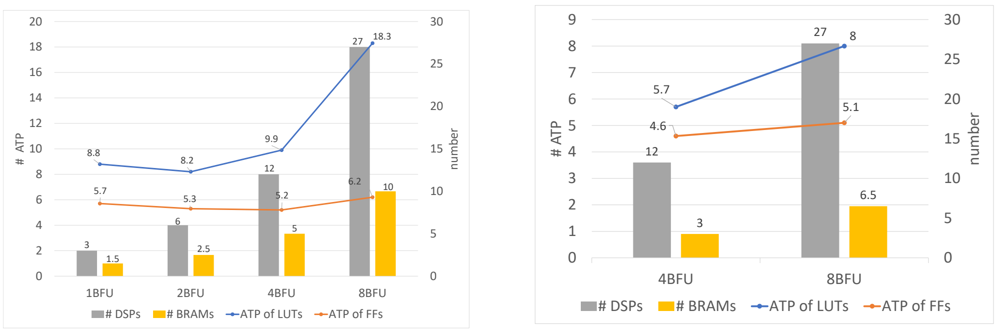


(d) FFs proportion in radix-4 NTT

Figure 11: The variation of LUTs/FFs proportion in radix-2 and radix-4 NTT.

Figure 11 indicates the variation of LUTs and FFs proportion in radix-2 and radix-4 NTT (N = 1024, 14-bit q) with different number of parallel butterfly units (parallel degrees) on the FPGA platform. The CTRL represents the control unit and the PT_AGU, TF_AGU are address generator units tailored for data points and twiddle factors, respectively. Based on these figures, we made two important observations. First, it is obvious that the resource proportion of AB and INN modules will become larger when increasing the number of parallel butterfly units in both radix-2 and radix-4 NTT. Moreover, the LUTs proportion of interconnect network accounts for the main part among these listed modules, especially in the case of larger parallel degrees. Second, the growth rate of LUTs proportion in radix-4 NTT is slower than that of radix-2 NTT. In other words, the growth rate of the number of fan-ins in INN of radix-4 NTT is almost fourfold lower than that of radix-2 NTT, which promotes the area-time efficiency in radix-4 NTT.

Figure 12 with dual coordinates depicts the comparison of ATP measured by LUTs/FFs and the number of DSPs/BRAMs between radix-2 and radix-4 NTT. The left coordinate indicates that the ATP of radix-4 NTT outperforms radix-2 NTT when configured with



(a) radix-2 NTT (b) radix-4 NTT

Figure 12: The variation of ATP in radix-2 and radix-4 NTT (N = 1024, 14-bit q).

the same number of butterfly units. This advantage is enlarged with the growth of parallel degree. When setting the number of butterfly units to 8, the ATP of LUT/FF in radix-4 NTT core is approximately $2.2\times/1.2\times$ less than the radix-2 NTT core. Moreover, the radix-4 NTT configured with 4 butterfly units achieves the best balance between area and time among these designs. The right coordinate indicates how the number of DSP and BRAM varies with the parallel degree. Since the radix-2 and radix-4 butterfly unit consume the same number of modular multiplication, the growth of DSP is identical as well. But the radix-4 NTT consumes less number of BRAMs. Table 4 in Appendix also presents the detailed resource usage broken down across different modules for radix-2 and radix-4 NTT cores with different number of parallel butterfly units. This table is made according to the resource utilization reported by Vivado 2020.2 on Windows 10.

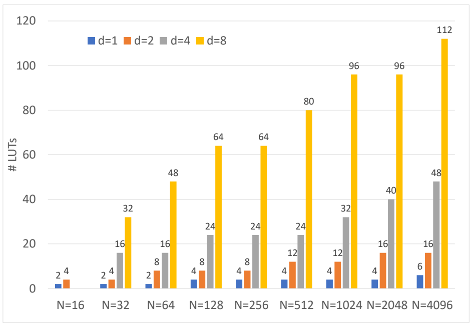
Table 2: Conflict-free memory mapping scheme comparisons with related work.

Scheme	**Radix Config.**	**BFU Config.**	**FFT Type**	**Storage**	**Architecture**	**Hardware Resource**
[Joh92]	$\surd$	$\times$	IP	N	Memory-based	XOR+Shift
[XMX17]	$\surd$	$\surd$	CG	$2N$		XOR+Shift
[RV08]	$\surd$	$\surd$	IP	$N\log_R N$		LUT+Shift
[BUC19]	$\times$	$\times$	CG	$2N$		XOR+Shift
[FLX20]	$\times$	$\surd$	SH	$2N$		Reorder+Shift
[RVM+14]	$\times$	$\surd$	IP	N		-
[XWXY17]	$\surd$	$\surd$	IP	$2N$	Memory-MDC-based	XOR+Shift
This Work	$\surd$	$\surd$	IP	N	Memory-based	XOR+Shift

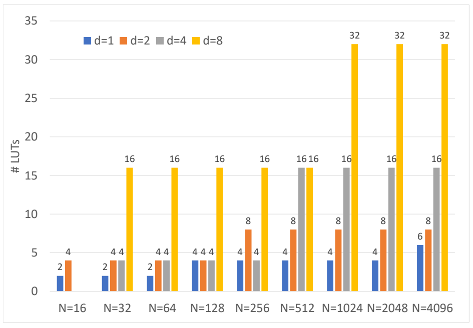
IP : in-place FFT. CG : constant geometry FFT. SH : Stockham FFT.

6.2 Comparisons with Related Work

Quantifying the difference of memory mapping scheme. The devised memory mapping scheme owns broader applicability but still maintaining low logic and memory hardware overhead, which is promoted by making full use of the toggling principle of address in every stage of in-place NTT. Table 2 lists the key comparison results between the related conflict-free memory mapping schemes and our proposed one. The concrete difference is mainly elaborated with five aspects as the following. **(1) Broader applicability.** It is observed that the memory mapping scheme in this work can support the configuration for arbitrary radix in-place NTT with valid number of parallel butterfly units. **(2) Smaller storage overhead.** [XMX17] proposes a memory mapping scheme for constant geometry radix-2^k FFT supporting multiple number of butterfly units. The dataflow of constant geometry FFT is much simpler than in-place FFT at the price of double memory overheads. Moreover, other schemes in Table 2 neither can backup the FFT radix/BFU configuration nor consume more hardware resource and memory footprint. **(3) Lower permutation complexity.** [XHY$^+$20] applies six types of permutation mode to guarantee the correct memory access. This method utilizes extra control hardware to configure the routing path of interconnect network according to the order of stage in NTT, which is fixed to suite the radix-2 NTT with 32 butterfly units as well. While the memory mapping scheme in this work only needs one type of memory mapping mechanism and thus reduces the redundant control overhead. **(4) Avoiding pipeline stalls.** The memory access scheme for radix-2 NTT in [RVM$^+$14] utilizes extra registers to buffer and synchronize the data points fetched at different operation moment. As a result, this method leads to pipeline stalls and more clock cycle consumption. While the proposed scheme carefully avoids this pipeline bubbles and thus improves the hardware utilization. **(5) Slightly lower LUT overhead.** [ZYC$^+$20] also presents a multi-bank NTT architecture based on the memory mapping scheme in [WHEW14]. This scheme is revised by additionally reordering the last loop of NTT to avoid the memory conflict mentioned in 2.2.2, which is limited to match radix-2 NTT as well. To be specific, the bank index is derived from a modular accumulation formula. To quantify and compare the resource consumption between [ZYC$^+$20] and the proposed one, we implement these two schemes on FPGA targeting at different vector length N and number of butterfly units d for radix-2 NTT. Figure 13 shows that the proposed one consumes slightly less amount of LUTs than [ZYC$^+$20] under the same vector length and parallel degree. Because the number of XOR chains in [ZYC$^+$20] will increase with the parallel degree while the proposed one is almost constant. Taking (N = 1024, d = 8) as an example, the bank index calculated by [ZYC$^+$20] is expressed in Verilog HDL style as: bank index $= a[9:8] + a[7:4] + a[3:0]$, which consists of four XOR chains corresponding to 4-bit width. Synthesis result shows that it consumes $6 \times 16 = 96$ LUTs for mapping the 16 logic addresses. While the bank index derived from the proposed formula in section 4 is expressed as: bank index $= sel == 0\ ?\ a[3:0] : \{\sim a[3], a[2:0]\}$, $sel = a[9] \oplus a[8] \oplus a[7] \oplus a[6] \oplus a[5] \oplus a[4]$, which consumes $2 \times 16 = 32$ LUTs. This advantage becomes even more prominent when increasing the vector length and parallel degree.



(a) Resource usages in [ZYC$^+$20].



(b) Resource usages in this work.

Figure 13: The comparison of radix-2 mapping scheme between [ZYC$^+$20] and this work.

Table 3: The comparison of NTT with related works.

Work	**BFU**	**Cycles**	**Freq. (MHz)**	**Time. (μs)**	**LUT/ ATP**	**FF/ ATP**	**DSP/ ATP**	**BRAM/ ATP**
parameters: N = 1024, fixed modulus q = 12289								
[ZYC+20]^A	2	2569*	244	10.5	847/8.9	375/3.9	2/21.1	6/63.2
[FS19]^A	1	10240	-	-	980/41.1	395/16.6	26/1091	2/83.9
[BUC19]^A	2	6155	-	-	7690/194	16/0.4	11/277.5	13/327.9
[JGCS19]^A	1	6206*	251	24.7	343/8.5	493/12.2	3/74.2	6/148.4
[XL20]^Z	4	2688	153	5.52	4823/26.6	2901/16.0	8/44.2	-
[KLC+17]^Z	4	2616*	150	17.44	2832/49.4	1381/24.1	8/139.5	10/174.4
parameters: N = 1024, tunable 14-bit modulus q								
[MKO+20a]^Z	32	200*	125	1.6	17188/27.5	-	96/153.6	48/76.8
[NDG19]^Z	2 × 2	2032	188	10.81	898/9.7	1117/12.1	4/43.24	10/108.1
This work^A	2 × 2	1308	213	6.1	1161/7.1	967/5.9	12/73.2	3/18.3
This work^V	2 × 2	1308	270	4.8	1196/5.7	969/4.6	12/57.6	3/14.4
	4 × 2	668	250	2.7	2953/8.0	1875/5.1	27/72.9	5.5/14.85
	1	5134	278	18.5	475/8.8	307/5.7	3/55.5	1.5/27.75
	2	2574	270	9.5	863/8.2	561/5.3	6/57	2.5/23.75
	4	1294	263	4.9	2011/9.9	1070/5.2	12/58.8	5/24.5
	8	654	227	2.9	6300/18.3	2124/6.2	27/78.3	10/29.0

* : The cycles for bit-reversed operation are not included.
A : Artix-7 FPGA platform. V : Virtex-7 FPGA platform. Z : Zynq-7000 FPGA platform.

The comparison of NTT with related works. As elaborated in section 6.1, the radix-4 NTT with two-layer butterfly units owns the best ATP performance. The radix-4 butterfly unit has a slightly larger pipeline latency than radix-2, which is constant and depends on the practical hardware implementation. But the heavier interconnect network of radix-2 NTT with 8 BFUs results in a larger frequency drop than radix-4 NTT with 4 × 2 BFUs. We mainly compare the implementation result of radix-4 NTT (2 × 2 BFUs) core with the state-of-the-art works as depicted in Table 3. [ZYC+20] proposes a radix-2 NTT design with two parallel butterfly units. It cannot naturally avoid the bit-reversed operation and consumes approximately 2× cycles as our design. The ATP of LUT is 1.5× greater than this work. Since it proposes a fixed architecture for a specific modulus 12289, the ATP of FF and DSP is smaller than ours. However, owing to reusing the twiddle factors in NTT and INTT, our architecture achieves almost 4.4× improvement in terms of BRAM ATP. [FS19] presents a radix-2 NTT design with a single butterfly unit. It consumes less number of BRAMs but the ATP of LUT, FF and DSP is approximately 7.2×, 3.6× and 18.9× as our work, respectively. [BUC19] puts forward a constant geometry radix-2 NTT with two parallel butterfly units as well. This design targets power-efficiency and supports multiple modulus. Therefore, it requires less FFs but the ATP of LUT, DSP and BRAM is almost 34.0×, 4.8× and 22.8× as our design, respectively. A fast and configurable radix-2 NTT is proposed in [JGCS19]. However, it needs extra cycles to execute the bit-reversed operation, pre-processing and post-processing. The ATP of LUT, FF, DSP and BRAM is almost 1.6×, 2.6×, 1.3× and 10.3× as our design, respectively. [XL20] presents a constant geometry radix-2 NTT architecture based on 4 parallel butterfly units.

However, it consumes more memory overhead and computational cycles. The ATP of LUT, FF is approximately 4.6×, 3.4× as our design. Because the modular multiplier is specific for the modulus 12289, thus it consumes less number of DSPs as our design. [KLC+17] also puts forward a parallel radix-2 NTT architecture with 4 butterfly units. However, it cannot avoid the pre-processing and post-processing, thus consuming extra more clock cycles. Its modular multiplier is based on a fixed Barrett reduction method for 12289. The ATP of LUT, FF, DSP and BRAM is almost 8.6×, 5.2×, 2.4×, and 12.1× as our work, respectively. [MKO+20a] proposes the highly parallel radix-2 NTT with 32 butterfly units, which results in less number of computational cycles. Nonetheless, the high fan-out and fan-in in this architecture will pull down the maximum frequency and it does not reuse the twiddle factors. Therefore, the ATP of LUT, DSP and BRAM is nearly 4.8×, 2.6× and 5.3× as our architecture. A two-layer radix-2 in-place NTT design based on high-level synthesis method is also presented in [NDG19]. But the tricks of reusing twiddle factor and radix-4 butterfly operator are not applied in this design. It also consumes more clock cycles to compute NTT. As a result, the ATP of LUT, FF and BRAM is approximately 1.7×, 2.6× and 7.5× as our work. We also synthesize and implement the radix-4 NTT core on Artix-7 (XC7Z020CLG484-3) FPGA platform considering the influence of FPGA type. As shown in Table 3, the implementation result of frequency is lower than the former one, but the resource usage is almost unchanged. The performance of the proposed radix-4 NTT core still has a clear advantage over the state-of-the-art works.

7 Conclusion

In this paper, we propose a scalable radix-2 and radix-4 NTT multiplication architecture based on an efficient memory mapping scheme. The detailed derivation process for bit-reversed-free radix-4 NTT and INTT with low complexity is provided. Both algorithm-level and architecture-level optimization techniques are applied to reduce the area overhead and memory footprint in our design. Through theoretical analysis and implementation results, we point out that the proposed radix-4 NTT kernel with the same number of parallel butterfly units outperforms the radix-2 one in terms of area-time performance. This advantage is enlarged when increasing the parallel degree. In addition, the proposed memory mapping scheme is able to support the parallelization of arbitrary radix NTT, which is more universal and efficient than other methods. In the future, we plan to integrate the NTT core into the PQC or FHE cryptosystem to improve the overall performance. It is interesting to consider the protective implementations for NTT core against the side channel attacks as well.

Acknowledgments

This work is supported in part by the National Key R&D Program of China (Grant No.2018YFB2202101), and in part by the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No.2018ZX01027101-002), and in part by the National Natural Science Foundation of China (Grant No.61804088). We thank Alan Szepieniec from Nervos for his valuable discussion about this work. We also thank the editors and reviewers for their thoughtful comments.

References

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for \{R, M\}lwe schemes. *IACR Cryptol. ePrint Arch.*, 2020:12, 2020.

[ACC$^+$21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial multiplication in NTRU prime comparison of optimization strategies on cortex-m4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):217–238, 2021.

[BGV93] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 1993.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of kyber on cortex-m4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):17–61, 2019.

[CG99] E. Chu and A. George. *Inside the FFT Black Box.* Inside the FFT Black Box, 1999.

[CHK$^+$21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for ntt-unfriendly rings new speed records for saber and NTRU on cortex-m4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[CMV$^+$15] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Chi-Wai Pao, and Ingrid Verbauwhede. High-speed polynomial multiplication architecture for ring-lwe and SHE cryptosystems. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 62-I(1):157–166, 2015.

[FBR$^+$21] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. Cryptology ePrint Archive, Report 2021/479, 2021. https://ia.cr/2021/479.

[FL19] Xiang Feng and Shuguo Li. Accelerating an FHE integer multiplier using negative wrapped convolution and ping-pong FFT. *IEEE Trans. Circuits Syst. II Express Briefs*, 66-II(1):121–125, 2019.

[FLX20] Xiang Feng, Shuguo Li, and Sufen Xu. Rlwe-oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm. *IEEE Trans. Circuits Syst. II Express Briefs*, 67-II(3):556–559, 2020.

[FS19] T. Fritzmann and J. Sepulveda. Efficient and flexible low-power ntt for lattice-based cryptography. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[Gen09] C. Gentry. Fully homomorphic encryption using ideal lattices. *Stoc*, 2009.

[GGMG13] Mario Garrido, Jesús Grajal, Miguel A. Sánchez Marcos, and Oscar Gustafsson. Pipelined radix-2^k feedforward FFT architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(1):23–32, 2013.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2013.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. *ACM*, 1966.

[HT96] Shousheng He and Mats Torkelson. A new approach to pipeline FFT processor. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*, pages 766–770. IEEE Computer Society, 1996.

[JGCS19] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. Spqcop: Side-channel protected post-quantum cryptoprocessor. *IACR Cryptol. ePrint Arch.*, 2019:765, 2019.

[Joh92] L. G. Johnson. Conflict free memory addressing for dedicated fft hardware. *IEEE Trans. on Circuit and Systems-II*, 39(5):312–316, 1992.

[KLC+17] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High performance post-quantum key exchange on fpgas. *IACR Cryptology ePrint Archive*, page 690, 2017.

[LNS20] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. Practical lattice-based zero-knowledge proofs for integer relations. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1051–1070. ACM, 2020.

[LSW01] Hsin-Fu Lo, Ming-Der Shieh, and Chien-Ming Wu. Design of an efficient FFT processor for DAB system. In *Proceedings of the 2001 International Symposium on Circuits and Systems, ISCAS 2001, Sydney, Australia, May 6-9, 2001*, pages 654–657. IEEE, 2001.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Jacob Alperin-Sheriff. Status report on the second round of the nist post-quantum cryptography standardization process, 2020-07-22 2020.

[MKO+20a] Ahmet Can Mert, Emre Karabulut, Erdinc Ozturk, Erkay Savas, and Aydin Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, pages 1–1, 2020.

[MKÖ+20b] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savas, Michela Becchi, and Aydin Aysu. A flexible and scalable NTT hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 346–351. IEEE, 2020.

[NDG19] Duc Tri Nguyen, Viet B. Dang, and Kris Gaj. A high-level synthesis approach to the software/hardware codesign of ntt-based post-quantum cryptography algorithms. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 371–374. IEEE, 2019.

[Nic71] Peter J. Nicholson. Algebraic theory of finite fourier transforms. *Journal of Computer and System Sciences*, 5(5):524–547, 1971.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 2015.

[RMD$^+$15] Stephen Richardson, Dejan Markovic, Andrew Danowitz, John Brunhaver, and Mark Horowitz. Building conflict-free FFT schedules. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 62-I(4):1146–1155, 2015.

[RV08] Dionysios I. Reisis and Nikolaos Vlassopoulos. Conflict-free parallel memory accessing techniques for FFT architectures. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 55-I(11):3438–3447, 2008.

[RVM$^+$14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[Sto06] H. S. Stone. R66-50 an algorithm for the machine calculation of complex fourier series. *IEEE Transactions on Electronic Computers*, EC-15(4):680–681, 2006.

[SYJ84] E.E. Swartzlander, W.K.W. Young, and S.J. Joseph. A radix 4 delay commutator for fast fourier transform processor implementation. *IEEE Journal of Solid-State Circuits*, 19(5):702–709, 1984.

[TCH19] Wei-Lun Tsai, Sau-Gee Chen, and Shen-Jui Huang. Reconfigurable radix-2 k$\times$ 3 feedforward fft architectures. In *IEEE International Symposium on Circuits and Systems, ISCAS 2019, Sapporo, Japan, May 26-29, 2019*, pages 1–5. IEEE, 2019.

[TJS03] J.H. Takala, T.S. Jarvinen, and H.T. Sorokin. Conflict-free parallel memory access scheme for fft processors. In *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.*, volume 4, pages IV–IV, 2003.

[WHEW14] Wei Wang, Xinming Huang, Niall Emmart, and Charles C. Weems. VLSI design of a large-number multiplier for fully homomorphic encryption. *IEEE Trans. Very Large Scale Integr. Syst.*, 22(9):1879–1887, 2014.

[XHY+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2672–2684, 2020.

[XL20] Yufei Xing and Shuguo Li. An efficient implementation of the newhope key exchange on fpgas. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 67-I(3):866–878, 2020.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of cca-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):328–356, 2021.

[XMX17] Qianjian Xing, Zhen-guo Ma, and Yingke Xu. A novel conflict-free parallel memory access scheme for FFT processors. *IEEE Trans. Circuits Syst. II Express Briefs*, 64-II(11):1347–1351, 2017.

[XWXY17] Kaifeng Xia, Bin Wu, Tao Xiong, and Tian-Chun Ye. A memory-based FFT processor design with generalized efficient conflict-free address schemes. *IEEE Trans. Very Large Scale Integr. Syst.*, 25(6):1919–1929, 2017.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of newhope-nist on FPGA using low-complexity NTT/INTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):49–72, 2020.

Appendix

The concrete dataflow of 27-point radix-3 DIT-NR NTT with three butterfly units is depicted in Figure 14. At each stage, 9 data points are accessed in parallel according to the growing group order. The lines denoted with red, yellow and blue color represent the parallel butterfly operations in the first round of each stage, respectively. The conflict-free memory mapping scheme proposed in section 4 is also valid for radix-3 in-place NTT. The detailed mapping process for 27-point radix-3 in-place DIT-NR NTT is depicted in Figure 15. Taking stage 0 ($p = 2$) as an example, the nine old addresses $\{0, 9, 18, 1, 10, 19, 2, 11, 20\}$ in group 0 round 0 ($G_{2,0,0}$) are mapped onto nine desired different bank indexes $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, respectively. By derivation, readers can find that other computation stages also hold valid as stage 0.

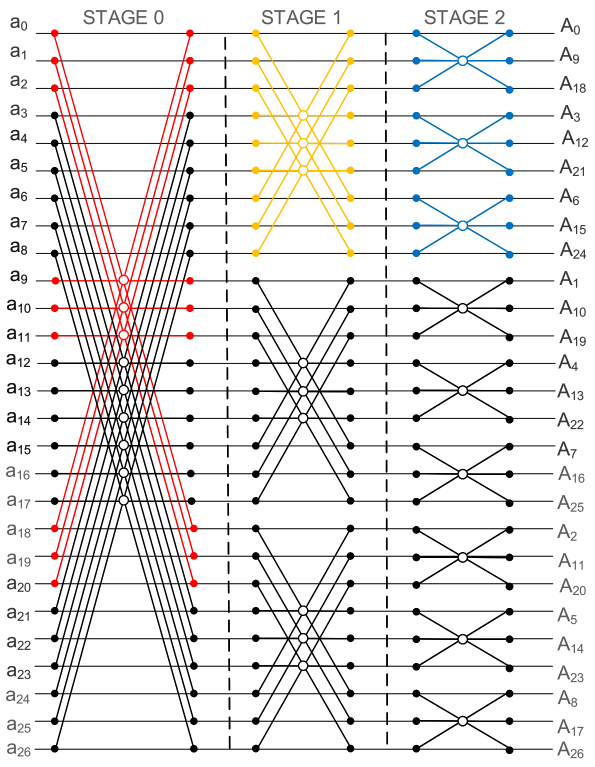
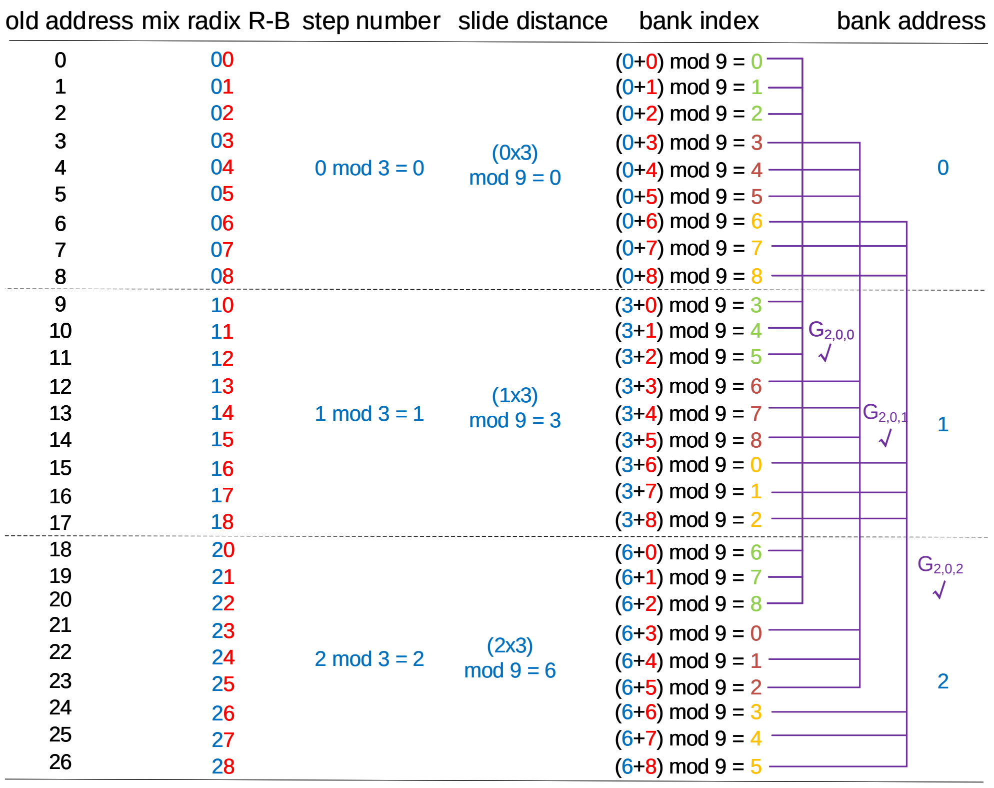


Figure 14: The dataflow of 27-point radix-3 DIT-NR NTT with d = 3.



old address	mix radix R-B	step number	slide distance	bank index	bank address
0	00			(0+0) mod 9 = 0	
1	01			(0+1) mod 9 = 1	
2	02			(0+2) mod 9 = 2	
3	03			(0+3) mod 9 = 3	
4	04	0 mod 3 = 0	(0x3) mod 9 = 0	(0+4) mod 9 = 4	0
5	05			(0+5) mod 9 = 5	
6	06			(0+6) mod 9 = 6	
7	07			(0+7) mod 9 = 7	
8	08			(0+8) mod 9 = 8	
9	10			(3+0) mod 9 = 3	
10	11			(3+1) mod 9 = 4	
11	12			(3+2) mod 9 = 5	
12	13			(3+3) mod 9 = 6	
13	14	1 mod 3 = 1	(1x3) mod 9 = 3	(3+4) mod 9 = 7	1
14	15			(3+5) mod 9 = 8	
15	16			(3+6) mod 9 = 0	
16	17			(3+7) mod 9 = 1	
17	18			(3+8) mod 9 = 2	
18	20			(6+0) mod 9 = 6	
19	21			(6+1) mod 9 = 7	
20	22			(6+2) mod 9 = 8	
21	23			(6+3) mod 9 = 0	
22	24	2 mod 3 = 2	(2x3) mod 9 = 6	(6+4) mod 9 = 1	2
23	25			(6+5) mod 9 = 2	
24	26			(6+6) mod 9 = 3	
25	27			(6+7) mod 9 = 4	
26	28			(6+8) mod 9 = 5	

Figure 15: The detailed memory mapping scheme for 27-point radix-3 NTT with d = 3.

Table 4: The resource usage broken down across different modules in radix-2 and radix-4 NTT cores. CTRL: control unit. PT_AGU: data point address generator. TF_AGU: twiddle factor address generator. MMU: memory mapping unit. INN: interconnect network. AB: Arbiter. BFU: butterfly unit. The resource utilization is reported by Vivado 2020.2 on Windows 10.

Modules	radix-2 NTT				radix-4 NTT	
	1 BFUs	2 BFUs	4 BFUs	8 BFUs	2 × 2 BFUs	4 × 2 BFUs
CTRL	103 LUTs/ 37 FFs	75 LUTs/ 35 FFs	89 LUTs/ 33 FFs	67 LUTs/ 31 FFs	56 LUTs/ 39 FFs	60 LUTs/ 35 FFs
PT_AGU	28 LUTs/ 0 FFs	38 LUTs/ 0 FFs	61 LUTs/ 0 FFs	67 LUTs/ 0 FFs	14 LUTs/ 0 FFs	61 LUTs/ 0 FFs
TF_AGU	82 LUTs/ 10 FFs	69 LUTs/ 10 FFs	56 LUTs/ 9 FFs	45 LUTs/ 9 FFs	30 LUTs/ 9 FFs	23 LUTs/ 8 FFs
MMU	4 LUTs/ 20 FFs	6 LUTs/ 40 FFs	13 LUTs/ 80 FFs	32 LUTs/ 160 FFs	10 LUTs/ 40 FFs	16 LUTs/ 80 FFs
INN	55 LUTs/ 11 FFs	242 LUTs/ 30 FFs	852 LUTs/ 78 FFs	3891 LUTs/ 198 FFs	224 LUTs/ 36 FFs	930 LUTs/ 84 FFs
AB	1 LUTs/ 0 FFs	12 LUTs/ 0 FFs	84 LUTs/ 0 FFs	429 LUTs/ 0 FFs	12 LUTs/ 0 FFs	87 LUTs/ 0 FFs
BFU	185 LUTs/ 188 FFs/ 3 DSPs	379 LUTs/ 376 FFs/ 6 DSPs	772 LUTs/ 752 FFs/ 12 DSPs	1561 LUTs/ 1504 FFs/ 24 DSPs	813 LUTs/ 768 FFs/ 12 DSPs	1677 LUTs/ 1536 FFs/ 24 DSPs
OTHER	19 LUTs/ 42 FFs	42 LUTs/ 70 FFs	84 LUTs/ 118 FFs	208 LUTs/ 222 FFs	31 LUTs/ 76 FFs	99 LUTs/ 132 FFs
TOTAL	477 LUTs/ 308 FFs/ 3 DSPs	863 LUTs/ 561 FFs/ 6 DSPs	2011 LUTs/ 1070 FFs/ 12 DSPs	6300 LUTs/ 2124 FFs/ 24 DSPs	1190 LUTs/ 968 FFs/ 12 DSPs	2953 LUTs/ 1875 FFs/ 24 DSPs