PipeNTT: A Pipelined Number Theoretic Transform Architecture

Zewen Ye, *Graduate Student Member, IEEE*, Ray C. C. Cheung, *Senior Member, IEEE*, and Kejie Huang, *Senior Member, IEEE*

***Abstract*—Polynomial multiplication is the key and time-consuming operation among various operators in Post-Quantum Cryptography (PQC), which aims to find quantum-resistant algorithms to prevent attacks launched by quantum computers. Number Theoretic Transform (NTT) is an efficient algorithm that can accelerate the polynomial multiplication from $\mathcal{O}(n^2)$ to $\mathcal{O}(nlog(n))$. In this brief, we present a pipelined NTT (PipeNTT) hardware architecture in FPGA to achieve high throughput with fewer hardware resources. The dataflow and the butterfly unit are optimized to minimize the latency. To fulfil the proposed dataflow, a Block RAM (BRAM) based reordering unit is designed to further reduce the hardware resource. Moreover, our architecture can also be applied to Inverse-NTT (INTT). Compared to state-of-the-art parallel designs, our design achieves a 30% lower area-time product with 3x less memory space requirement.**

***Index Terms*—Post-quantum cryptography (PQC), number theoretic transform (NTT), pipelined design, FPGA.**

I. Introduction

POST-QUANTUM Cryptography (PQC) [1] is under intensive development to protect their data against quantum computers. However, many PQC schemes suffer from high computational complexity due to the polynomial multiplications. For example, the polynomial multiplications occupy 78% of computation time in Kyber [2]. The polynomial multiplication can be accelerated by Number Theoretic Transform (NTT), which is a variant Fast Fourier Transform (FFT) operating on a finite field. Most of the conventional FPGA implementations of NTT are using parallel computing architectures at the cost of high hardware resources. Some hardware designs are isolated design instances with fixed parameters, or a very narrow range of parameters [3]. What's worse, as the level of parallelism increases, the control logic circuits become very complex to deal with data conflict, resulting in large memory space and high power consumption [4]–[7], which is not suitable for the power-constrained end devices for Internet-of-Thing (IoT) applications. Pipelined FFT scheme is an efficient scheme to reduce hardware resource. However, conventional pipelined FFT designs such as Single-path Delay Feedback (SDF) and Multi-path Delay Commutator (MDC) [8] have various limitations, which is inefficient for NTT. The SDF design uses feedback circuits to reorder data while the MDC design inserts delays between butterfly units. Both of the designs use butterfly units with two input and output ports, leading to halving the utilization ratio of butterfly units and the long latency. The work [9] presented a pipelined FFT design with specific I/O order to reduce the latency. However, the resource consumption of registers increases rapidly as the bit length of numbers and the number of stages increase.

To reduce the hardware resource and memory consumption, we propose a novel Pipelined NTT (PipeNTT) hardware with advantage of consecutive NTT operations in FPGA for a wide range of parameter sets. First, we design a Block RAM (BRAM) based reordering circuit to reduce the hardware cost in data reordering. Second, based on Decimation-In-Time (DIT) NTT algorithm, we design a butterfly unit with a higher utilization rate. Unlike the dataflow in [9], our dataflow is the reversal form to take advantage of our butterfly unit. We also adopt the k-RED algorithm [10] in our design. Third, our architecture can also be used in Decimation-In-Frequency (DIF) Inverse-NTT (INTT). The DIT and DIF are the form of Cooley-Tukey and Gentleman-Sande, respectively. Experimental results show that our pipelined NTT design consumes 3x less memory and achieves 30% less area-time product.

The rest of this brief is organized as follows. Section II reviews the NTT algorithm and conventional parallel NTT designs. Section III illustrates the details of the PipeNTT, including hardware architecture and dataflow scheme. Section IV describes how our architecture can be applied to INTT. Section V shows the experimental results and the comparisons with prior works. Finally, Section VI summarizes the main contributions of this brief.

II. Background

A. Number Theoretic Transform

NTT is a Discrete Fourier Transform defined over the polynomial ring $\mathbb{Z}_q[x]/\phi(x)$ where $\phi(x)$ is usually defined

Manuscript received 6 April 2022; revised 2 June 2022; accepted 16 June 2022. Date of publication 21 June 2022; date of current version 26 September 2022. This work was supported in part by the National Natural Science Foundation of China under Grant U19B2043, and in part by the National Key Research and Development Program of China under Grant 2020AAA0109002. This brief was recommended by Associate Editor C.-A. Shen. *(Corresponding author: Kejie Huang.)*

Zewen Ye is with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310058, China, and also with the Department of Electrical Engineering, City University of Hong Kong, Hong Kong (e-mail: lucas.zw.ye@zju.edu.cn).

Ray C. C. Cheung is with the Department of Electrical Engineering, City University of Hong Kong, Hong Kong (e-mail: r.cheung@cityu.edu.hk).

Kejie Huang is with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310058, China (e-mail: huangkejie@zju.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSII.2022.3184703.

Digital Object Identifier 10.1109/TCSII.2022.3184703

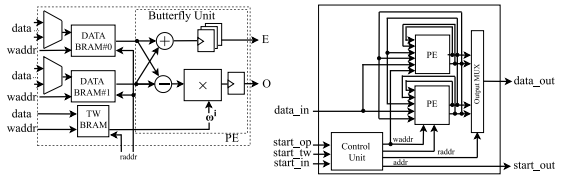

Fig. 1. The parallel NTT design [5].

as $x^n + 1$ for polynomial multiplication. NTT takes a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$ as input and returns the output $A(x) = \sum_{i=0}^{n-1} A_i x^i$, where

$$A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \ mod \ q \tag{1}$$

where ω is the twiddle factor as well as the primitive n-th root-of-unity and q is the modulus. For INTT, the twiddle factor ω is replaced to multiplicative inverse ω^{-1}, and additional final scaling N^{-1} is required:

$$a_i = N^{-1} \sum_{j=0}^{n-1} A_j \omega^{-ij} \ mod \ q. \tag{2}$$

B. Prior Parallel NTT Implementations

Fig. 1 shows the conventional parallel NTT design [5]. In each PE, there are two BRAMs for coefficients and one BRAM for twiddle factors. Each PE receives two inputs from BRAMs and returns two outputs. The outputs of one PE are connected to all PEs to broadcast the coefficients and write back to the corresponding BRAMs. The final outputs are reordered by the Output MUX. The number of multipliers, adders, subtractors, and BRAMs are proportional to the number of PEs, but the complexity of control logic increases exponentially with the growth of the number of parallel PEs.

III. Pipelined NTT Architecture

To improve the computing efficiency and hardware utilization, we propose a pipelined NTT architecture, as illustrated in Fig. 2. PipeNTT has $N = log_2(n)$ stages and each stage requires one butterfly unit (BFU) to perform butterfly operation. Reordering Unit (RU) is used to reorganize the order of data and Twiddle Factor RAM (TW RAM) is used to provide the twiddle factors for each stage. There are two types of RU, BRAM-based RU (BRU) and Flip-Flop-based RU (FRU). Besides, the BRAMs of BRUs and TW RAMs of different stages have different RAM capacities. The control logic, TW RAMs, and BRAMs in each stage are all independent. To handle all stages concurrently, each control logic generates read and write address of BRAM, read address of TW RAM, and control signal of Mux independently.

A. Butterfly Unit

The BFU circuit shown in Fig. 2 utilizes the DIT iterative NTT scheme in which modular multiplication is performed before modular addition and subtraction. The BFU is fully pipelined and the workflow is illustrated in Fig. 3,where ω is the twiddle factor, and a and b are the input coefficients. The input coefficients are taken from BRAM or the preceding stage, and the output coefficients will be stored to BRAM or sent to the succeeding stage. It takes $l + 2$ cycles to perform the modular multiplication of b and ω, and the temporary data $b \cdot \omega$ is latched by a register to wait for data a to be ready for modular addition or subtraction. The two output data (a' and b') are generated sequentially. To fully utilize hardware resource, we present our optimized modular operations in the following.

Algorithm 1: k^l-RED Modular Multiplication

Input: $x = (x_{t-1} \cdots x_1 x_0)_2$, $y = (y_{t-1} \cdots y_1 y_0)_2$,
$q = (q_{t-1} \cdots q_1 q_0)_2 = k \cdot 2^m + 1$, $l = \lceil t/m \rceil$
Output: $r = xyk^l \ mod \ q$

```
C ← x · y
for i from 0 to l − 1 do
    C ← k · C[m − 1 : 0] − C[2t − 1 − i · m : m]
end
if C > q then
    C ← C − q
else if C < 0 then
    C ← C + q
end
return C
```

1) Modular Addition and Subtraction: Modular addition and subtraction Algorithms from [11] are used in our design to take full advantage of hardware. Fig. 4 shows the details of the modular addition and subtraction hardware. In our design, the modular addition and modular subtraction are completed in one clock cycle.

2) Modular Multiplication: The modular multiplication hardware is the most time-consuming part in NTT. Barrett reduction [12] and Montgomery reduction [13] are the most commonly used modular reduction algorithms. Both of them require several multiplications and additions. However, the k-RED algorithm [10] is a more efficient choice thanks to the properties of modulus q. The modulus q used in NTT should satisfy $q \equiv 1 (\text{mod } 2n)$ and therefore $q = k \cdot 2^m + 1$. k^2-RED, a variant of K-RED algorithm, can get the modular reduction result directly [14]. In our design, we extend the k^2-RED to k^l-RED for different NTT parameters as described in Algorithm 1 where q is represented by m and k, t is the bit-length of input number and l is the number of loop. It can be seen that we can not directly get the result $xy \bmod q$. However, since the multiplicand is the twiddle factor which is pre-computed, we can pre-computed $\omega \cdot k^l$ and store it to memory instead of ω. We split the pipeline according to the number of multiplications, so the total stages of modular multiplication are $l+2$. In this way, the critical path of our design is multiplication.

B. Dataflow and Reordering Unit

1) Dataflow: In our design, the data sequences required for different stages are different. Our proposed dataflow scheme to schedule different stages is shown in Fig. 5 ($n = 16$ in this example), which is modified from [9]. Because BFUs compute the multiplications first, we put the coefficients for multiplications in the first place to reduce delay. The delay of

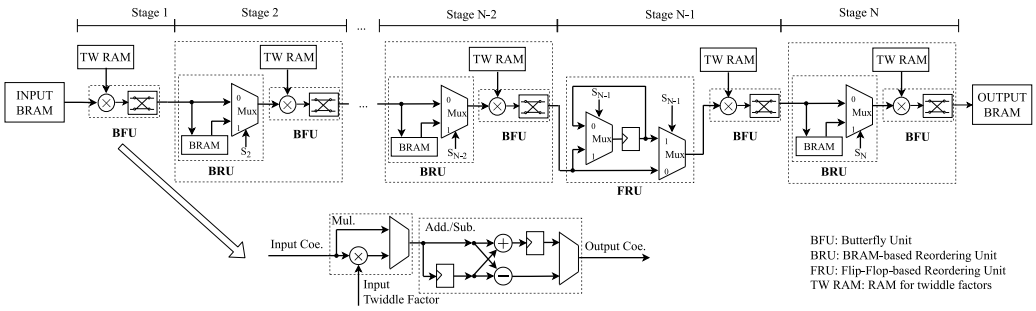

Fig. 2. The architecture of the proposed PipeNTT.

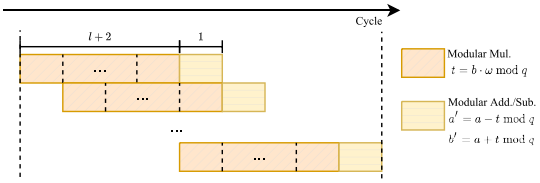

Fig. 3. The pipelined flow of BFU.

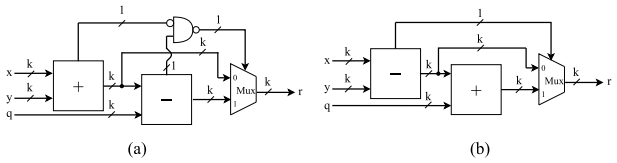

Fig. 4. The hardware of (a) modular addition and (b) modular subtraction.

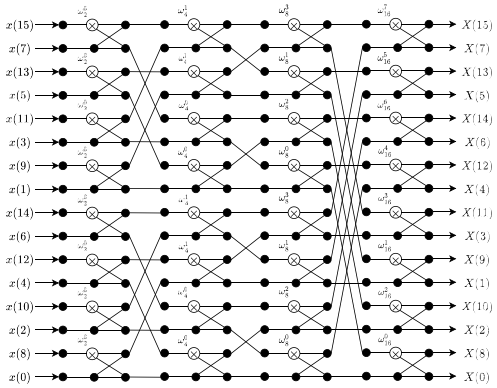

Fig. 5. Our modified dataflow of 16-pt NTT.

different stages between BFUs are inserted by RUs to ensure the data are reorganized to the desired sequences.

We denote the input data sequence as $\alpha(0), \alpha(1), \ldots, \alpha(n-1)$ and the output data sequence as $\beta(0), \beta(1), \ldots, \beta(n-1)$. The input coefficients taken from BRAM are:

$$\alpha(i) = \begin{cases} x(\frac{n}{2} - i + (\frac{n}{2} - 1)(i \bmod 2 = 0)), & 0 \le i < \frac{n}{2} \\ x(n - 1 - i + (\frac{n}{2} - 1)(i \bmod 2 = 0))), & \frac{n}{2} \le i < n \end{cases} \quad (3)$$

The output coefficients written to BRAM are:

$$\beta(i) = \begin{cases} X(BR(n - 2i - (i \bmod 2 = 0))), & 0 \le i < \frac{n}{2} \\ X(BR(2n - 2i - 2 - (i \bmod 2 = 0))), & \frac{n}{2} \le i < n \end{cases} \quad (4)$$

where **BR** represents the **BitReverse** function.

2) Reordering Unit: To compute NTT correctly using our PipeNTT, the stages must wait for the required data and store the temporary data in BRAM. As illustrated in Fig. 5, $x(15)$ and $x(7)$ are the first and second inputs to stage 1, respectively. In stage 2, the RU provides 3 cycles delay to reorganize the data sequence. Generally, the delay of stage i is

$$d_i = \begin{cases} 2^{N-i} - 1, & 2 \le i < N \\ 2^{N-1} - 1, & i = N \end{cases} \quad (5)$$

Hence, the total delay of our PipeNTT is:

$$\sum_{i=2}^{N-1}(2^{N-i} - 1) + 2^{N-1} - 1 = n - N - 1 = n - log_2(n) - 1. \quad (6)$$

To achieve the desired order, we use RU for permutations of serial data. There are two types of RUs in Fig. 2. For $d_i = 1$, we use the FRU presented in [9] to provide one cycle delay. To provide different delays, our general BRU will read the parameter d_i. There are two possible cases in our BRU:

- When the data is required for the current stage, the control signal of MUX is 0 and the data can go through RU.
- When the data is unused and the required data is stored in BRAM, the control signal of MUX becomes 1. The unused data will be written to BRAM and the required data will be read from BRAM using the write address and read address generated by the control logic, respectively.

The write address and read address of BRAM, and the control signal of MUX are generated using additional f-bit counters counting from $(0 \cdots 0)_2$ to $(1 \cdots 1)_2$, where m is the bit-length of BRAM address. The details are provided in the following:

- **Write Address** is obtained by a f-bit counter and the counter starts when RU is enabled.
- **Read Address** is obtained by another f-bit counter that is enabled after d_i cycles when RU is enabled.
- **Control Signal of MUX** is: S_j = (NOT c_j) OR c_0 for stage j, where c_j and c_0 come from the **Write Address Counter** $(c_{n-1}, \ldots, c_0)_2$.

To avoid data conflict during reordering, the depth of RU BRAM should be large enough. Since the number of data stored in RU BRAM is d_i at most, we choose the depth to be $2(d_i + 1)$ which is power-of-2 to avoid data conflict. Note that some RU BRAMs with very small depth will be synthesized into LUTs in FPGA. The memory consumption of BRAMs for all RUs is: $\sum_{i=1}^{N-1} 2(d_i + 1) = \sum_{i=1}^{N-1} 2(2^{N-i} - 1 + 1) = 2n - 8$.

TABLE I
IMPLEMENTATION RESULTS AND COMPARISON TO PRIOR WORKS

Work	Platform	n, $log_2(q)$	LUT/FF/DSP/BRAM	Area[a]	Cycles	Clock (MHz)	Avg. Time[b]	TP[c]	LUT × T	Area × T
[15][d]	Zynq-7000	512, 14	741/330/2/5	2441	1289	245	5.3	1352	3927	12937
This	Virtex-7		**2178/1694/8/3**	**3878**	**1074**	**234**	**2.3**	**3117**	**5009**	**8919**
[15][d]	Zynq-7000	1024, 14	847/375/2/6	2847	2569	244	10.5	1365	8894	29894
[14][d]	Artix-7		798/715/4/2	1798	1591	234	6.8	2108	5426	12226
[16]	Artix-7		4938/2882/8/-	-	1280	152	8.4	1703	41578	-
This	Virtex-7		**2151/1475/9/2.5**	**3801**	**2104**	**232**	**4.5**	**3186**	**9680**	**17105**
[17][d]	Artix-7	256, 12	2543/792/4/9	5643	232	182	1.3	2363	3306	7336
This	Virtex-7		**1362/1036/7/0.5**	**2212**	**556**	**235**	**1.2**	2560	**1634**	**2654**
[5]	Virtex-7	256, 16	7400/5000/24/24	17000	160	147	1.1	3724	8140	18700
This			**2294/1645/7/3**	**3894**	**556**	**228**	**1.2**	**3413**	**2753**	**4673**
[5]	Virtex-7	512, 16	8100/5200/24/24	17700	345	141	2.4	3413	19440	42480
This			**2482/1876/8/4**	**4482**	**1074**	**227**	**2.5**	**3277**	**6205**	**11205**
[18]	UltraScale+	1024, 16	3140/3007/58/29	17640	6212	183	33.9	483	106446	597996
This	Virtex-7		**2225/1624/9/3**	**4025**	**2104**	**230**	**4.5**	**3640**	**10013**	**18113**
[5]	Virtex-7	1024, 28	132K/59K/448/96	205.6K	250	125	2	14336	118K	411.2K
			16K/14K/56/24	28.8K	490	125	3.9	7352	62K	112.3K
This			**3.4K/3.1K/63/6**	**11.5K**	**2114**	**175**	**6**	**4779**	**20.4K**	**69K**
[5]	Virtex-7	4096, 60	338K/138K/1984/768	767.8K	972	125	7	35109	2366K	5375K
			22K/17K/248/96	75.6K	3276	125	26	9452	572K	1966K
This			**17K/11K/286/24.5**	**52.9K**	**8284**	**150**	**27.5**	**8937**	**467.5K**	**1455K**

[a] The Area is measured by #LUTs + 100 × #DSPs + 300 × #BRAMs; [b] Average Latency is measured by running 100 times NTT; [c] Throughput is the number of processing bits per second; [d] The modular reduction is optimized for specific parameter q without DSPs.

3) Data Management: The TW RAM is used to provide the twiddle factors for each stage. Each stage requires $\frac{n}{2}$ twiddle factors. However, as shown in Fig. 5, there is only ω_2^0 in stage 1 and only ω_4^0, ω_4^1 in stage 2. In this case, we reuse some twiddle factors to reduce the size of TW RAM. Generally, for any stage i, the number of twiddle factors is 2^{i-1}. The order of stage $i (1 \leq i < N)$ for reading twiddle factors is: $\omega_{2^i}^{BR(0)}, \omega_{2^i}^{BR(1)}, \ldots, \omega_{2^i}^{BR(2^{i-1}-1)}$. While for stage N, the order of twiddle factors is: $\omega_{2^i}^{BR(0)}, \omega_{2^i}^{BR(2)}, \ldots, \omega_{2^i}^{BR(2^{i-1}-2)}, \omega_{2^i}^{BR(1)}, \ldots, \omega_{2^i}^{BR(2^{i-1}-1)}$. In total, the memory space required for all twiddle factors of all stages is: $\sum_{i=2}^{N} 2^{i-1} = n - 2$.

The BRAMs used in the input and output store the coefficients, so the total memory consumption of input and output is $2n$. In total, the overall memory consumption including RU BRAM, TW RAM, INPUT BRAM, and OUTPUT BRAM is: $(2n - 8) + (n - 2) + 2n = 5n - 10$.

IV. INTT IMPLEMENTATION

The twiddle factors and the final scaling of INTT are the main difference between NTT and INTT. Our design can configure the TW RAM to change the twiddle factors. The final scaling operations require additional multipliers and computing latency. In our design, the idea of merging the final scaling into each stage is used [15]. Additional "1/2" operations that can be computed in one cycle are added to each stage so INTT needs #stages more cycles than NTT. Because INTT is always computed after NTT in many cases, we consider the DIF INTT which can use the output data of pipelined NTT without any permutations. The INTT dataflow shown in Fig. 6 is symmetric with NTT dataflow in Fig. 5. The RUs used in INTT are the same as our pipelined NTT design while the delays of each stage are the reversal, i.e., the delays of stage 2, 3 and 4 in the 16-pt INTT are the same as the delays of stage 4, 3 and 2 in the 16-pt NTT, respectively.

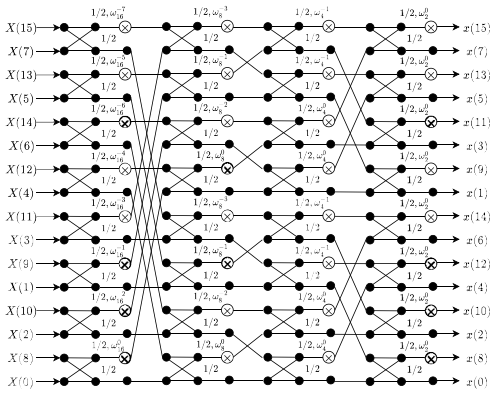

Fig. 6. The dataflow of 16-pt INTT.

V. EXPERIMENTS AND RESULTS

We present a detailed evaluation of the proposed PipeNTT architecture. All the designs are implemented using Verilog on Virtex-7 XC7VX485T FPGA and Xilinx Vivado 2017.2 is used to perform synthesis, place, and route with default settings.

Our flexible and scalable architecture gives users a wide range of design options based on the application requirement and resource availability. We compare our designs with existing implementations on FPGA [5], [14]–[18] in terms of the consumed resources, latency, throughput, and area-time product (ATP), as shown in Table I. The works [14]–[16] are system-level designs so we take their original experimental

TABLE II
THE BREAKDOWN OF THE RESOURCE UTILIZATION OF OUR DESIGN

n, $log_2(q)$	LUT	FF	DSP	BRAM
256, 12	1362 (0.45%)	1036 (0.17%)	7 (0.29%)	0.5 (0.05%)
512, 14	2178 (0.72%)	1694 (0.28%)	8 (0.29%)	3 (0.29%)
1024, 14	2151 (0.71%)	1475 (0.24%)	9 (0.32%)	2.5 (0.24%)
256, 16	2294 (0.75%)	1645 (0.27%)	7 (0.25%)	3 (0.29%)
512, 16	2482 (0.81%)	1876 (0.3%)	8 (0.29%)	4 (0.39%)
1024, 16	2225 (0.73%)	1624 (0.27%)	9 (0.32%)	3 (0.29%)
1024, 28	3400 (1.1%)	3100 (0.5%)	63 (2.25%)	6 (0.58%)
4096, 60	17K (5.6%)	11K (1.8%)	286 (10.2%)	24.5 (2.37%)

results of NTT and the works [5], [17], [18] are pure NTT designs. To provide fair comparisons, we normalize the LUT, DSP, and BRAM to the total area. The DSP is approximately equal to 100 LUTs according to [19]. Using the same approximation method, one BRAM can be approximately equal to 300 LUTs. Because our pipelined design targets on consecutive NTT operations, we provide the average latency which is the average running time in 100 times consecutive NTT operations. The breakdown of the resource utilization of our design is shown in Table II.

The works [14], [15], [17] use fixed modulus q in their design so there is no multiplication in their modular reduction unit. In this case, their design don't have flexibility for different modulus q. Since our design can be used for any modulus q with the same bit-length, we consume more resources in DSP. The work [16] is a parallel design with 4 NTT cores. As can be seen, our design outperforms it with more than 3x less LUT-time product and nearly 2x throughput. The work [18] is also a pipelined design. But due to the inefficient modular reduction algorithm it used, our design has advantage in both area and time.

As the control logic and memory management of our design are simpler than parallel multi-core design, our design consumes fewer resources than [5] especially in LUTs, FFs, and BRAMs. For small parameters ($n = 256, 512$, $log_2(q) = 16$), our designs achieve similar latency with 3x less ATP and 6-8x fewer BRAMs than the parallel design in [5]. For the case of large parameters, our proposed scheme has more than 30% less ATP and 3x fewer BRAMs than the multi-core parallel designs.

Our design supports a wide range of parameter sets and can be applied to PQC schemes such as CRYSTALS-DILITHIUM, NewHope, and qTESLA and homomorphic encryption such as BFV and CKKS. For CRYSTALS-KYBER that data sequence is split into two parts with half-length to perform NTT [2], we can perform NTT operations on the two sets of sequences separately. For polynomial multiplication, the pre-processing and post-processing of NTT and INTT can be done by only changing the data in TW RAMs [15].

VI. CONCLUSION

In this brief, we have presented an optimized pipelined NTT FPGA implementation. Our PipeNTT architecture is a scalable and flexible design that can be applied to a variety of NTT parameters. The data management and the control logic of the overall design are much simpler than parallel NTT design. Our PipeNTT consumes less area and power consumption, which can play an important role in IoT and edge computing applications.

REFERENCES

[1] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Comput. Surveys*, vol. 51, no. 6, pp. 1–41, 2019.

[2] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "A monolithic hardware implementation of kyber: Comparing apples to apples in PQC candidates," in *Proc. Int. Conf. Cryptol. Inf. Security Latin America*, 2021, pp. 108–126.

[3] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 3–16, Jan. 2017.

[4] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020.

[5] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi, and A. Aysu, "A flexible and scalable NTT hardware: Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2020, pp. 346–351.

[6] Y. Zhang, C. Wang, D. E. S. Kundi, A. Khalid, M. O'Neill, and W. Liu, "An efficient and parallel R-LWE cryptoprocessor," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 886–890, May 2020.

[7] X. Feng, S. Li, and S. Xu, "RLWE-oriented high-speed polynomial multiplier utilizing multi-lane Stockham NTT algorithm," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 3, pp. 556–559, Mar. 2020.

[8] X. Liu, F. Yu, and Z.-K. Wang, "A pipelined architecture for normal I/O order FFT," *J. Zhejiang Univ. Sci. C*, vol. 12, no. 1, pp. 76–82, 2011.

[9] M. Garrido, S.-J. Huang, S.-G. Chen, and O. Gustafsson, "The serial commutator FFT," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 63, no. 10, pp. 974–978, Oct. 2016.

[10] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Proc. Int. Conf. Cryptol. Netw. Security*, 2016, pp. 124–139.

[11] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 4, pp. 17–61, 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8344

[12] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Conf. Theory Appl. Cryptograph. Techn.*, 1986, pp. 311–323.

[13] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[14] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "High-speed NTT-based polynomial multiplication accelerator for post-quantum cryptography," in *Proc. IEEE 28th Symp. Comput. Arith. (ARITH)*, 2021, pp. 94–101.

[15] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 2, pp. 49–72, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8544

[16] Y. Xing and S. Li, "An efficient implementation of the NewHope key exchange on FPGAs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 3, pp. 866–878, Mar. 2020.

[17] F. Yarman, A. C. Mert, E. Öztürk, and E. Savaş, "A hardware accelerator for polynomial multiplication operation of crystals-kyber PQC scheme," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2021, pp. 1020–1025.

[18] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt, "Exploring energy efficient quantum-resistant signal processing using array processors," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, 2020, pp. 1539–1543.

[19] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Cryptographic accelerators for digital signature based on Ed25519," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 7, pp. 1297–1305, Jul. 2021.