Scalable and Parallel Optimization of the Number Theoretic Transform Based on FPGA

Bin Li, Yunfei Yan, Yuanxin Wei, and Heru Han

***Abstract*— In lattice-based postquantum cryptography (PQC), polynomial multiplication is complex and time-consuming, which affects the overall computational efficiency. In addition, the parameters of different lattice-based algorithms require different number theoretic transform (NTT) structures, which limits the versatility of hardware design. To this end, this article proposes scalable and parallel optimization of the NTT based on a field-programmable gate array (FPGA). By analyzing the algorithm flow of the NTT, inverse NTT (INTT), and pointwise multiplication (PWM), an FPGA loosely coupled structure is designed, which can be used to place butterfly units of multiple pipelines in parallel and supports various modulo operations of a polynomial. In addition, to improve computing efficiency and scalability, key algorithm modules such as multipliers and modular reduction are deeply optimized. Moreover, the storage optimization of multiple RAM channels is carried out, and the alternate access control of data and the multiplexing of RAM resources reduce resource consumption and improve data access efficiency. For the SHA-3 algorithm, the scalable Keccak algorithm is implemented in a serial–parallel hybrid manner and supports multiple hash modes. Finally, taking the Dilithium algorithm as an example, through the parallelization of SHA-3 and NTT, the calculation cycle of key generation, signature, and verification is shortened. The experimental results and analysis show that the scheme in this article shortens the NTT calculation period while ensuring a high frequency, and the calculation time is significantly better than that of other schemes. Furthermore, it can support the optimized parallelization of multiple moduli and give full play to the computing advantages of an FPGA.**

***Index Terms*— Butterfly arithmetic, field-programmable gate array (FPGA), number theoretic transform (NTT), polynomial multiplication, SHA-3.**

I. Introduction

INFORMATION security is the foundation of the development of modern society. It realizes the confidentiality of communication transmission, the integrity of data interactions, and the authentication of trusted entities and data sources with a cryptographic system that is recognized to be sufficiently secure. Most traditional cryptographic systems use Rivest–Shamir–Adleman (RSA) and elliptic curve cryptography (ECC) to realize key exchange, digital signature authentication, and identity authentication, and their security depends on the integer decomposition problem and the discrete logarithm problem. However, with the development of quantum computing, the traditional public key cryptosystem is facing serious security problems [1], causing RSA and ECC to be broken in polynomial time. Therefore, to resist quantum attacks, postquantum cryptography (PQC) has emerged, which is an important research direction now.

At present, PQC schemes mainly include lattice-, code-, multivariate-, hash- [2], and isogeny-based [3] schemes. When PQC replaces ECC and RSA, every security application, from smartphones to block chains, will be affected [4], [5]. Some resource-constrained scenarios [6] focus on compact and energy-efficient PQC designs, while others pursue high throughput and better performance. The cryptographic algorithm based on the lattice problem has the advantages of high encryption efficiency, fast calculation speeds, and various functions and is easy to integrate and chip, making it a very promising cryptographic scheme [7]. Following the collection and screening of PQC algorithms by the National Institute of Standards and Technology (NIST), four algorithms were finally identified as PQC standard algorithms, including three lattice-based algorithms—CRYSTALS-Kyber, CRYSTALS-Dilithium, and the FALCON cryptography schemes. Although PQC has been studied to varying degrees in terms of public key encryption, key exchange, and digital signatures, it is still in its infancy.

In lattice-based cryptography (LBC), polynomial multiplication is the key operation, which affects the overall computational efficiency of the cryptographic algorithm. Among existing schemes, the number theoretic transform (NTT) can quickly realize the multiplication of polynomials [8], [9] and its computational complexity is $O(n \log n)$. However, the NTT control logic contains a multilayer loop-nested operation structure, and the access scheduling of polynomial coefficients is extremely complicated. Second, the butterfly operation is iteratively executed in NTT many times and includes various operations such as modular addition and subtraction, modular multiplication, and modular reduction, which greatly affect the delay of hardware computing. Finally, there are many parameters in the current LBC, and different parameters are required to design different NTT structures, which limits the versatility of hardware design. Therefore, how to use the reconfigurable feature of a field-programmable gate

Manuscript received 23 April 2023; revised 15 July 2023 and 15 August 2023; accepted 3 September 2023. Date of publication 28 September 2023; date of current version 23 January 2024. This work was supported in part by the Key Scientific and Technological Project of Henan Province under Grant 232102211055 and in part by the Henan Key Laboratory of Network Cryptography Technology under Grant LNCT2022-A14. *(Corresponding author: Bin Li.)*

Bin Li is with the School of Computer and Artificial Intelligence, Zhengzhou University, Zhengzhou, Henan 450001, China, and also with the Henan Key Laboratory of Network Cryptography Technology, Zhengzhou, Henan 450001, China (e-mail: cctvlibin@163.com).

Yunfei Yan, Yuanxin Wei, and Heru Han are with the School of Computer and Artificial Intelligence, Zhengzhou University, Zhengzhou 450001, China.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2023.3312423.

Digital Object Identifier 10.1109/TVLSI.2023.3312423

array (FPGA) to realize efficient and scalable NTT hardware structures with reconfigurable configurations urgently needs to be solved.

Nowadays, for the realization of a lattice-based cryptosystem, it is still necessary to take into account the execution efficiency and scalability of the hardware, and the hardware needs as many parallel structures as possible to achieve faster operation speeds. Therefore, with respect to the high-efficiency application requirements of PQC, this article carries out the optimized acceleration and scalable design of the NTT algorithm. First, the algorithm flow of NTT, inverse NTT (INTT), and pointwise multiplication (PWM) is analyzed. Then, the overall architecture of an FPGA is given, and multiple butterfly computing units are used for parallel computing to improve the computing efficiency. Second, the optimization design of the pipeline butterfly operation unit is given in detail, and the memory access is optimized with multichannel RAM to reduce the calculation delay. Finally, the K^2-RED module reduction algorithm and SHA-3 algorithm are optimized and designed to adapt to various parameters and calculation modes, and NTT and SHA-3 are parallelized to improve scalability.

The main innovations of this article are as follows.

1) Configure multiple pipeline butterfly units in parallel on the FPGA with a loosely coupled architecture and use multiple RAMs to control data access to improve computing efficiency. The design supports various polynomial operations such as NTT, INTT, PWM, modular addition, modular subtraction, and modular multiplication.
2) A butterfly unit is designed with a configurable Karatsuba–Ofman Algorithm (KOA) multiplier and K^2-RED modular reduction, which reduces the consumption of DSP and supports various parameter sets of (n, q) for multiple LBC algorithms.
3) Multi-RAM access is optimized, the coefficients and twiddle factors of the polynomial are rearranged and stored independently, and data conflicts are prevented by reading first and then writing, realizing continuous reading and writing of data, satisfying the computational requirements of pipelined butterfly units.
4) The high-speed and efficient SHA-3 algorithm is optimized to support multiple calculation modes of SHA3-256/512 and SHAKE128/256. Through parameter configuration, it can meet the generation of pseudorandom numbers and sampling values of multiple LBC algorithms.
5) Using multiple NTT and SHA-3 in parallel shortens the calculation cycle of polynomial multiplication and sampling generation and uses first-in–first-out (FIFO) as the interface to simplify data communication and improve scalability.

The remainder of this article is organized as follows. Section II introduces the NTT algorithm and its hardware implementation. Section III gives the design and optimization process of the scalable NTT hardware structure and SHA-3 algorithm and then carries out parallel processing. Section IV presents the implementation results and analysis. Finally, Section V summarizes and concludes the article.

Algorithm 1 NTT Forward Transform

Input: $a(x) \in R_q, \omega, n, q$
Output: $A(x) = \mathrm{NTT}(a) \in R_q$

```
r = 1
for (i = (log2 n − 1) to 0) do
    m = 2^i
    for (k = 0 to (n/2m − 1)) do
        w = ω^br(r) mod q
        r = r + 1
        for (j = 0 to (m − 1)) do
            u = a[2km + j]
            v = a[2km + j + m] · w mod q
            a[2km + j] = (u + v) mod q
            a[2km + j + m] = (u − v) mod q
        end for
    end for
end for
return a
```

II. Related Work

A. Number Theoretic Transform

An integer ring is represented by $\mathbb{Z}$, $\mathbb{Z}_q$ represents the quotient ring of the integer ring modulo q, and $R_q = \mathbb{Z}_q[x]/\phi(x)$ represents the integer coefficient polynomial ring of the modulo polynomial $\phi(x)$, where $\phi(x)$ is an irreducible polynomial. In most cases, $\phi(x) = x^n + 1$, then $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ represents a polynomial ring whose degree is at most $n - 1$. NTT is a linear orthogonal transform defined on the ring $\mathbb{Z}_q$ modulo q. For a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$ of degree n on the ring R_q, where $a_i \in \mathbb{Z}_q$, let ω be the nth-order primitive unit root modulo q, satisfying $\omega^n \equiv 1 (\mathrm{mod} q)$ and $\omega^i \neq 1 (\mathrm{mod} q), \forall i < n$. Then, the NTT transformation of $a(x)$ is $A_i = \mathrm{NTT}_\omega(a)_i = \sum_{j=0}^{n-1} a_j \omega^{ij} (\mathrm{mod} q), i = 0, 1, 2, \ldots, n - 1$. Among them, A_i is the coefficient of polynomial $A(x) = \sum_{i=0}^{n-1} A_i x^i$ of degree n, and $A_i \in \mathbb{Z}_q$.

Similarly, for the NTT inverse transformation, INTT, it is necessary to calculate the modular inverse $\omega^{-1} (\mathrm{mod} q)$ and $n^{-1} (\mathrm{mod} q)$ on $\mathbb{Z}_q$, where ω^{-1} and n^{-1} satisfy $\omega \times \omega^{-1} \equiv 1 \bmod q$ and $n \times n^{-1} \equiv 1 \bmod q$, respectively, and they are calculated according to the following formula: $a_i = \mathrm{INTT}_{\omega^{-1}}(A)_i = n^{-1} \sum_{j=0}^{n-1} A_j \omega^{-ij} (\mathrm{mod} q), i = 0, 1, 2, \ldots, n - 1$. Then, there is $C = \mathrm{INTT}(\mathrm{NTT}(A) \odot \mathrm{NTT}(B))$ for NTT-based polynomial multiplication, where $\odot$ represents the multiplication PWM of polynomial coefficients.

Specifically, the flow of the NTT algorithms is shown in Algorithm 1, where $\mathrm{br}(r)$ represents the bit reverse operation of $\log_2 n$ bits on r. The algorithm flow of INTT is similar to that of NTT. It takes $A(x)$ as input and $a(x)$ as output, and calculates $u = A[2km+j]$, $v = A[2km+j+m]$, $A[2km+j] = ((u+v)/2) \bmod q$, and $A[2km+j+m] = ((u-v)w/2) \bmod q$ in the loop process.

B. Hardware-Related Implementation

At present, research on LBC mainly focuses on the NTT and INTT algorithms [10], which mainly include optimization of butterfly operations, storage access, and instruction sets.

In terms of algorithm optimization, Yaman et al. [11] accelerated the calculation of NTT/INTT with 16 butterfly computing units, but the calculation of its modular reduction is more complicated. Huang et al. [12] optimized the Kyber algorithm on Xilinx FPGA and designed the NTT module in the form of a pipeline, but the NTT calculation cycle is longer. Mert et al. [13] designed a butterfly computing unit based on Montgomery modular multiplication, which can be configured through parameters to complete NTT calculations in parallel with multiple processing elements (PEs). In addition, Mert et al. [14] used Montgomery modular multiplication to optimize the NTT in units at the word level and realized software and hardware collaboration through PCIe by DMA. However, as the prime modulus increases in these schemes, the calculation cycle of the Montgomery modular multiplication also increases correspondingly. Multiplication operations are still required, resulting in a decrease in computational efficiency. Meanwhile, these schemes do not support polynomial addition, subtraction, and accumulation after the multiplication of coefficients. Xing and Li [15] implemented the complete Kyber algorithm on Xilinx Artix-7, including the calculation of SHA-3 and NTT/INTT, but the scheme is still based on serial execution, and the calculation of NTT requires 512 clock cycles. Ricci et al. [16] implemented the NTT operation of the Kyber on Xilinx Virtex UltraScale+ XCVU7P high-performance FPGA with a frequency of 637 MHz, but its calculation cycle is still as high as 405 clocks. Furthermore, Ricci et al. [17] implemented the Dilithium digital signature algorithm on the FPGA and accelerated the calculation by inputting two sets of data each time with four butterfly computing units. Ma et al. [18] carried out a parallel implementation of NTT with the Kyber algorithm, which can complete the calculation within 287 clock cycles, reducing the iteration time. Zhou et al. [19] proposed a software–hardware coordination scheme for the Dilithium algorithm, which is called by the software in the form of an instruction set. Khatib et al. [20] optimized the SIKE algorithm through the Montgomery modular multiplication algorithm. Beckwith et al. [21] implemented a high-performance Dilithium algorithm through the 2×2 NTT structure. Zhang et al. [22] optimized two polynomial multiplication methods—schoolbook and NTT—and further accelerated the NTT calculation process with radix-4. Hu et al. [23] presented a "decompose-and-reduce" modular multiplication algorithm (DARM), considering primes with the form of $q = 2^{2n} - \delta$ and $\delta < 2^{n-2}$. However, the prime modulus form of the NIST PQC algorithm is $q = k \times 2^n + 1$, where k is odd and $k < 2^n$. Therefore, the DARM algorithm cannot be used in these PQC algorithms. In addition, multiplication operations are still required in the DARM algorithm. Azarderakhsh et al. [24] implemented the supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol on an FPGA, which is 1.5 times faster than that of software.

In terms of storage optimization, Chen et al. [25] used dual-column sequential storage to solve RAM read and write conflicts and optimized the butterfly operation with pipeline technology to improve computing efficiency. Li et al. [26] proposed a MeNTT accelerator, which optimizes the design of the SRAM array through a bit-serial modular arithmetic protocol and mapping strategy and improves the data throughput. Nejatollahi et al. [27] proposed a CryptoPIM accelerator solution with high-throughput in-memory computing, which supports up to 32 k NTT polynomial multiplication operations.

In terms of instruction set optimization, Seiler [28] optimized the NTT algorithm with the AVX2 instruction set and achieved a 6.3-fold performance improvement on the Skylake processor. Fritzmann et al. [29] designed RISQ-V based on the RISC-V architecture to accelerate the calculation of the LBC algorithm. Chung et al. [30] optimized NTT polynomial multiplication on Cortex-M4 and AVX2 processors with 32-bit and 16-bit processors, respectively. Bos et al. [31] implemented the first-, second-, and third-order mask designs of the Kyber algorithm on ARM Cortex-M0+ and Cortex-M4F, which improved the security of the algorithm during execution. Agrawal et al. [32] implemented a parameterized register transfer level (RTL) code library, which can provide n-point NTT operations and be used for fast polynomial multiplication.

In terms of high-level comprehensive language design, Zijlstra et al. [33] used high-level synthesis (HLS) to optimize and implement LBC algorithms based on learning with errors (LWE), ring-LWE (RLWE), and module-LWE (MLWE) and made a detailed comparison of various implementations in terms of resource and time consumption. Basu et al. [34] used HLS to implement various PQC algorithms and conducted a comparative analysis between resources and performance. Haleplidis et al. [35] implemented NTT and INTT parallel computing with the OpenCL heterogeneous programming language and tested and evaluated it on a GPU to improve the computing efficiency.

In terms of scalability, Chen et al. [36] proposed a radix-2 and radix-4 NTT polynomial multiplication architecture and used radix-4 butterfly unit operations to reduce the amount of computation, but the overall structure is more complex and lacks certain flexibility. Duong-Ngoc and Lee [37] proposed a configurable mixed-base NTT structure. By decomposing n points into two parts, that is, radix-2^{k_1} and radix-2^{k_2}, for operation, it supports polynomial multiplication of various degrees, but the structure is fixed with four PWM modules and lacks certain scalability. Derya et al. [38] proposed a configurable NTT polynomial multiplication accelerator, which realized the operation of any 32-bit modulus with the Montgomery reduction algorithm and can be configured with multiple butterfly units for parallel computing, which has good scalability. However, this solution does not support any modulus exceeding 32 bits, and there is still space for optimization in the butterfly operation structure.

In terms of antiside channel attacks [39], including fault attacks [40], [41], power analysis attacks, and timing attacks [42], the most common countermeasure is to

add concealment or masking techniques to the algorithm. Dubrova et al. [43] demonstrated side-channel attacks on up to the fifth-order masked implementations of CRYSTALS-Kyber in ARM Cortex-M4 CPU. Berzati et al. [44] designed a template attack on Dilithium that can efficiently predict whether a given coefficient in one coordinate of this vector is zero or not. Mozaffari-Kermani and Azarderakhsh [45] proposed reliable and error detection hash trees for hash-based postquantum signatures. Moreover, Kermani and Azarderakhsh [46] presented an error detection scheme for the AES-GCM algorithm. Aghaie et al. [47] proposed fault diagnosis schemes for an efficient lightweight block cipher to ensure a high level of security with low hardware overhead incorporated for error detection.

In summary, there are many optimization methods for the current LBC algorithm, and the polynomial power, modulus, and bit width are different. However, most schemes design the NTT structure with a fixed polynomial power and coefficient size and a fixed number of PEs. This limits the implementation of NTT for different platforms, different areas, and different performance requirements. Obviously, to improve the hardware computing speed and scope of application of the NTT algorithm, one option is to decompose the critical path, optimize it in depth, and improve the operation efficiency; the other is to increase the degree of parallelism, including the parallelism between modules and the ability to process data at the same time; and the third is to parameterize the module to meet the diversity of different LBC parameters. For this reason, this article starts from the calculation stages of NTT, INTT, and PWM and designs reconfigurable modules for the butterfly calculation, data access, control logic, and storage structure to improve scalability while ensuring calculation efficiency.

III. Algorithm Optimization Design

A. Overall Structure of NTT

The NTT algorithm includes multiple layers of loop nesting, involving a large number of modular arithmetic operations. The coefficient storage and scheduling are complex, and the bandwidth requirements are high, which affects the overall efficiency and operation delay of the cryptographic algorithm. To improve the overall flexibility and scalability of polynomial multiplication hardware, several butterfly operation units are placed in the hardware in a software-defined manner, and each butterfly operation unit can be executed independently. Second, multiple RAM channels are placed to access the polynomial coefficients and the results of each iteration. The twiddle factor ω is calculated in advance by means of precomputation and stored in the form of RAM. Finally, the control logic can control the input and output of the butterfly operation unit, the control of the RAM access address, and the reading control of the parameter ω. The overall structure of NTT is shown in Fig. 1. It is interconnected in a loosely coupled manner and can flexibly complete NTT, INTT, PWM, modular addition, modular subtraction, and modular multiplication.

B. Butterfly Operation Unit

The butterfly operation unit is the core of the NTT, INTT, and PWM calculations. The pipeline structure is used for

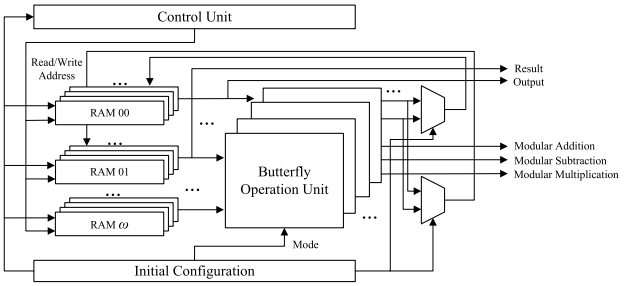

Fig. 1. Overall structure of the NTT algorithm.

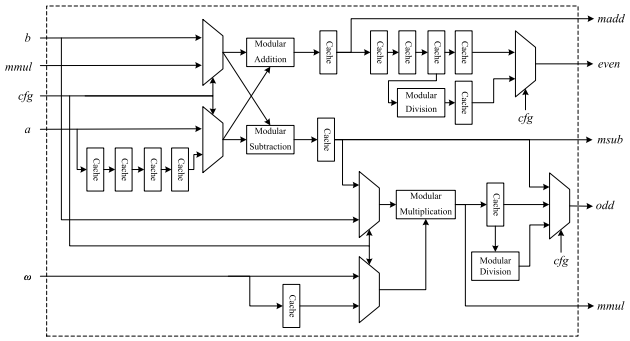

Fig. 2. Butterfly operation unit.

implementation, and according to the control signal, different calculation processes of NTT, INTT, and PWM are selected. The structure of the butterfly operation unit is shown in Fig. 2.

NTT calculates modular multiplication first, then modular addition and subtraction, whereas INTT first calculates modular addition and subtraction and then modular multiplication, hence the calculation orders of the two transforms are different. Therefore, a plurality of cache registers are inserted in the butterfly operation unit to cache the intermediate results and perform arbitration output according to the control signal cfg. Moreover, the logical level of the butterfly operation is reduced by caching to avoid a large path delay. Specifically, for NTT, even and odd are the calculation results of $a + b\omega(\mathrm{mod}q)$ and $a - b\omega(\mathrm{mod}q)$, respectively. For INTT, even and odd are the calculation results of $(a + b)/2(\mathrm{mod}q)$ and $(a - b)\omega/2(\mathrm{mod}q)$, respectively. Second, for the PWM algorithm, it is necessary to perform modular multiplication first and then the modular addition. Therefore, a and ω are used as multipliers, and b is used as the cumulative sum. After the intermediate modular multiplication result mmul is obtained, it is added to cached b, and the modular addition operation is performed. In this way, the coefficients corresponding to the polynomial matrix and the polynomial vector and the intermediate accumulation results are input in sequence to obtain the coefficients of the final polynomial vector. Moreover, this operation is applicable to polynomial vector operations that multiply first and then add. Finally, the butterfly operation unit can directly output the operation results madd, msub, and mmul of modular addition, modular subtraction, and modular multiplication, which is convenient for various operations between polynomial coefficients.

In the butterfly operation unit, there are already schemes [48] for modular division, modular addition, and modular subtraction. Modular multiplication is a key

Algorithm 2 KOA Multiplication

Input: X, Y, n
Output: P
1: $\{X_{\mathrm{H}}, X_{\mathrm{L}}\} = X$
2: $\{Y_{\mathrm{H}}, Y_{\mathrm{L}}\} = Y$
3: $P_0 = X_{\mathrm{H}} \times Y_{\mathrm{H}}$
4: $P_1 = X_{\mathrm{L}} \times Y_{\mathrm{L}}$
5: $P_2 = (X_{\mathrm{H}} + X_{\mathrm{L}})(Y_{\mathrm{H}} + Y_{\mathrm{L}})$
6: $P_3 = P_2 - P_1 - P_0$
7: $P = (P_0 << n) + (P_3 << (n/2)) + P_1$

component, which includes two parts: a multiplier and a modular reduction, as described below.

1) KOA Multiplier: The core idea of the KOA [49] algorithm is "divide and conquer," which uses a recursive method to decompose a complex multiplication operation into multiple simple multiplication operations, which is faster and more efficient than traditional calculations. For two n-digit numbers, if they are directly multiplied, the complexity is $O(n^2)$, and the KOA algorithm can reduce the complexity to $O(n^{\log_2 3})$. For an n-digit X, it can be denoted as: $X = (\underbrace{x_{n-1}, x_{n-2}, \ldots, x_{n/2}}_{X_{\mathrm{H}}}, \underbrace{x_{n/2-1}, \ldots, x_0}_{X_{\mathrm{L}}})$, where $x_i \in \{0, 1\}$, $0 \le i < n$. Then, the numbers X and Y can be equivalently expressed as: $X = X_{\mathrm{H}} \times 2^{n/2} + X_{\mathrm{L}}$, $Y = Y_{\mathrm{H}} \times 2^{n/2} + Y_{\mathrm{L}}$.

Therefore, $P = X \times Y$ can be calculated by the following formula: $P = (X_{\mathrm{H}} \times 2^{n/2} + X_{\mathrm{L}}) \times (Y_{\mathrm{H}} \times 2^{n/2} + Y_{\mathrm{L}}) = (X_{\mathrm{H}}Y_{\mathrm{H}}) \times 2^n + (X_{\mathrm{H}}Y_{\mathrm{L}} + X_{\mathrm{L}}Y_{\mathrm{H}}) \times 2^{n/2} + X_{\mathrm{L}}Y_{\mathrm{L}}$, and $X_{\mathrm{H}}Y_{\mathrm{L}} + X_{\mathrm{L}}Y_{\mathrm{H}} = (X_{\mathrm{H}} + X_{\mathrm{L}})(Y_{\mathrm{H}} + Y_{\mathrm{L}}) - X_{\mathrm{H}}Y_{\mathrm{H}} - X_{\mathrm{L}}Y_{\mathrm{L}}$; thus, the whole algorithm only needs to calculate the multiplications $X_{\mathrm{H}}Y_{\mathrm{H}}$, $X_{\mathrm{L}}Y_{\mathrm{L}}$, and $(X_{\mathrm{H}} + X_{\mathrm{L}})(Y_{\mathrm{H}} + Y_{\mathrm{L}})$ to reduce the complexity.

From the above analysis, it can be seen that using the KOA algorithm to calculate the multiplication of two large numbers only needs three multiplication operations and six addition and subtraction operations, which can effectively reduce the circuit's complexity. Especially when n is larger, the KOA multiplier can optimize the use of circuit resources more obviously. To improve the scope of application of the KOA algorithm, it is configured with n as a parameter, and the specific implementation is shown in Algorithm 2.

In Algorithm 2, the multiplication of Lines 3–5 is implemented by calling the underlying DSP of the FPGA. In addition, to increase the operating frequency of KOA, registers are used to cache intermediate results and form a three-stage pipeline. The circuit corresponding to the KOA multiplier is shown in Fig. 3.

2) K^2-RED Module Reduction: Longa and Naehrig [50] proposed a modulus reduction algorithm suitable for modulus $q = k \times 2^m + 1$, including the K-RED and K-RED-2x methods, and for any positive number D returning $E \equiv kD \bmod q$ and $E \equiv k^2 D \bmod q$ as two results. On this basis, Bisheh-Bisheh-Niasar et al. [51] proposed a K^2-RED modular reduction algorithm that, through two consecutive K-RED operations, returns $E \equiv k^2 D \bmod q$. When $q = 3329$, K^2-RED reduces one shift and addition operation compared with KRED-2x.

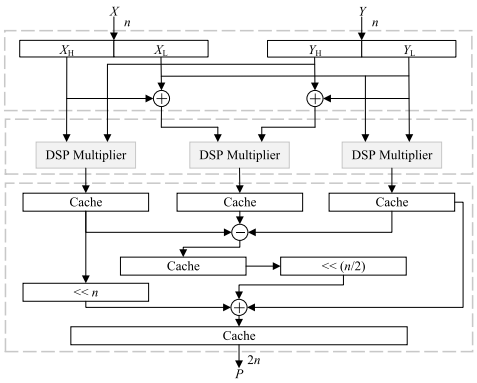

Fig. 3. KOA multiplier structure.

Algorithm 3 K-RED Algorithm

Input: positive integer $D \in [0, q^2)$
Output: $E = k \times D \bmod q$
1: $D_0 = D \bmod 2^m$
2: $D_1 = D/2^m$
3: return $k \times D_0 - D_1$

When the modulus form is $q = k \times 2^m + 1$, $k \times 2^m \equiv -1 \bmod q$, and $2^m > k$, then, for the modulo q operation of the product D of a and b, D can be expressed as $D = D_0 + 2^p D_1$ to obtain the K-RED algorithm, as shown in Algorithm 3.

In Algorithm 3, the value range of D' is $(-kq,\ kq)$, so it is necessary to add or subtract multiples of q to D' to ensure that the result falls into $[0, q)$. On this basis, the K^2-RED algorithm is formed by calling the K-RED algorithm twice consecutively, which narrows the value range of the result.

To improve the scope of application of modulus reduction, the K^2-RED algorithm is optimized and improved, and various modulus reduction operations can be completed according to the parameter configuration. First, when the modulus $q = k \times 2^m + 1$ is determined, then k and m are determined, and the bit width of the input D is IN_{size}, and the bit width of the output result E is $IN_{\mathrm{size}}/2$. Second, the K^2-RED algorithm replaces the multiplication operation by shifting and adding operations, so k can be expanded into binary form, and specific shift parameters can be obtained. Then, according to the result of the second K-RED operation, it is necessary to judge its high-order part and add a certain difference to ensure the correctness of the operation result. Finally, the output of the K^2-RED algorithm is a signed number. After testing, for the commonly used modulus q, the result range is $(-q,\ 3q)$. Therefore, it is also necessary to judge the final result E to determine whether to add or subtract the modulus q several times and to ensure that the final result of the reduction is nonnegative. The details are shown in Algorithm 4.

In Algorithm 4, the formulas of D' and D'' corresponding to the specific parameter k are shown in Table I.

The hardware structure corresponding to the scalable K^2-RED algorithm is shown in Fig. 4.

Since the K^2-RED algorithm scales the result by k^2 times each time, each ω in the NTT and INTT calculations is multiplied by the modular inverse $k^{-2} \bmod q$ during precomputation, and the final correct result can be obtained.

TABLE I
D' AND D'' FORMULAS CORRESPONDING TO k

q	k	D'	D''
12289	3	$D' = (D_0 << 1) + D_0 - D_1$	$D'' = (D'_0 << 1) + D'_0 - D'_1$
7340033	7	$D' = (D_0 << 2) + (D_0 << 1) + D_0 - D_1$	$D'' = (D'_0 << 2) + (D'_0 << 1) + D'_0 - D'_1$
5767169	11	$D' = (D_0 << 3) + (D_0 << 1) + D_0 - D_1$	$D'' = (D'_0 << 3) + (D'_0 << 1) + D'_0 - D'_1$
3329	13	$D' = (D_0 << 3) + (D_0 << 2) + D_0 - D_1$	$D'' = (D'_0 << 3) + (D'_0 << 2) + D'_0 - D'_1$
7681	15	$D' = (D_0 << 3) + (D_0 << 2)$ $+(D_0 << 1) + D_0 - D_1$	$D'' = (D'_0 << 3) + (D'_0 << 2)$ $+(D'_0 << 1) + D'_0 - D'_1$
8380417	1023	$D' = (D_0 << 10) - D_0 - D_1$	$D'' = (D'_0 << 10) - D'_0 - D'_1$

Algorithm 4 K^2-RED Algorithm

Input: positive integer $D \in [0, q^2)$, q, k, m, IN_{size}
Output: $E = k^2 \times D \bmod q$

$D_0 = D[m-1:0]$
$D_1 = D[IN_{\text{size}} - 1 : m]$
Calculate D' according to k
$D'_0 = D'[m-1:0]$
$D'_1 = D'[IN_{\text{size}} - m - 1 : m]$
Calculate D'' according to k
$D_2 = (D'_1[(IN_{\text{size}} - 2m - 1) : (N_{\text{size}} - 2m)/2] == 0)?\ D'' : D'' + (1 << (N_{\text{size}} - 2m))$
$D_3 = D_2 - q$
$D_4 = D_3 - q$
$D_5 = D_2 + q$
$D_6 = (D_2[IN_{\text{size}}/2] \ \&\ D_3[IN_{\text{size}}/2])?\ D_5 : ((D_4[IN_{\text{size}}/2] == 0)?\ D_4 : D_3)$
$E = (D_2[IN_{\text{size}}/2] == 0)?\ ((D_3[IN_{\text{size}}/2] == 0)?\ D_3 : D_2) : D_6$
return E

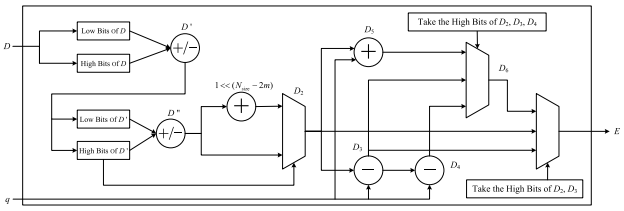

Fig. 4. Scalable K^2-RED algorithm hardware structure.

Furthermore, when calculating PWM, since the coefficients of polynomials A and B are multiplied and scaled by k^2 times, it is still necessary to multiply by the modular inverse k^{-4} mod q to obtain the correct result of PWM.

C. Multi-RAM Optimization

For NTT with N points, a total of $\log_2 N$ rounds of butterfly operations need to be performed, and each round of access to RAM requires a unique address to access the parameters. The butterfly operation process of eight points is given in Fig. 5.

Fig. 5 shows that in the first stage, the four sets of coefficient pairs (0, 4), (1, 5), (2, 6), and (3, 7) perform butterfly operations. Therefore, it is necessary to read each group of coefficient pairs from RAM within one clock cycle. Then, two RAMs (RAM0 and RAM1) can be used to store coefficients 0, 1, 2, and 3 and 4, 5, 6, and 7, respectively, for implementation. Additionally, the coefficient pairs change at each stage. In the second stage, the four sets of coefficient pairs involved in

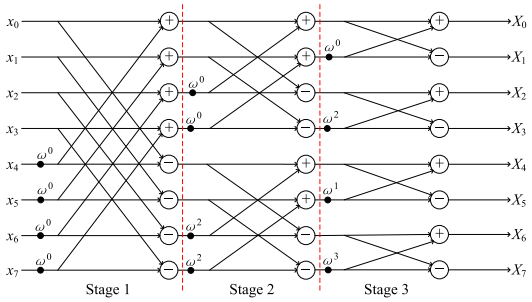

Fig. 5. Butterfly operation and RAM access of eight points.

the butterfly operation are (0, 2), (1, 3), (4, 6), and (5, 7). Therefore, the output results of the first stage (0, 4) and (1, 5) need to be stored in RAM0, and the results of (2, 6) and (3, 7) should be stored in RAM1. In this way, it can be ensured that in the second stage, the corresponding coefficient pair is still read from RAM0 and RAM1 within one clock cycle. Similarly, for the calculation in the third stage, it is necessary to adjust the storage of the RAM in the second stage.

1) Multiple RAM Storage: For the convenience of data access, this article adopts three RAMs corresponding to a butterfly operation unit, among which the first two RAMs store the coefficients of polynomials, and the third RAM stores the twiddle factors. Then, for the parallel NTT operation with M butterfly units, the whole structure needs $3M$ RAMs in total.

Specifically, taking NTT with 256 points and four butterfly units as an example, a total of 12 RAMs are needed, as shown in Figs. 6 and 7, which show the first round of initialization parameters and the storage situation of the twiddle factor RAM. Among them, the first eight RAMs are divided into four groups, each RAM corresponds to 32 addresses, there are $32 \times 8 = 256$ parameters in total, and 128 groups of coefficient pairs are stored in each group of RAMs in a certain order. When executing NTT, the coefficient pairs composed of the ith and $(i + 128)$th $(0 \le i < 128)$ polynomial parameters that perform butterfly operations are selected from each of the four groups of RAMs in the same clock cycle to perform four-way parallel operations. The last four RAMs are used to store twiddle factors, each RAM corresponds to 127 addresses, and there are $127 \times 4 = 508$ parameters in total. Since the parameters of the twiddle factor are fixed and some parameters are reused, after preprocessing, they are stored in RAM in a certain order to reduce resource occupation. For example, the 0th address of RAM is ω_{128}, which is used for the first stage of the butterfly operation; the ω_{64} and ω_{192} of the first and

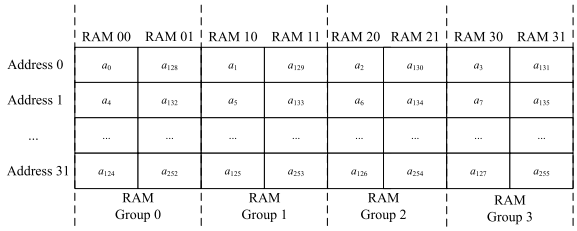

Fig. 6. Storage of polynomial coefficients.

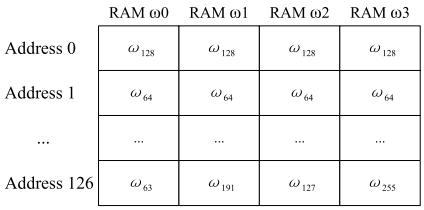

Fig. 7. Storage of twiddle factors.

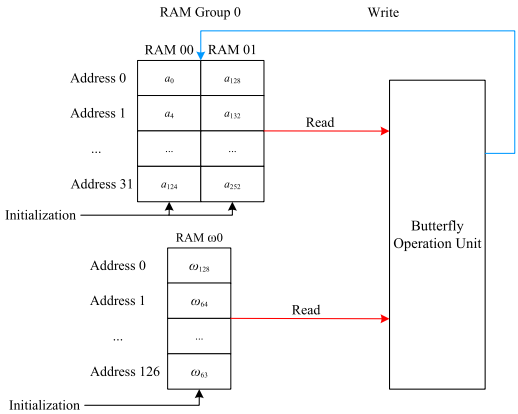

Fig. 8. RAM process.

second addresses are used for the second stage of the butterfly operation, and so on.

Taking group 0 RAM as an example, the RAM process is shown in Fig. 8. First, RAM is initialized, and polynomial coefficients and twiddle factors are written into RAM. Then, the coefficient pairs and twiddle factors are read from RAM and sent to the butterfly unit for calculation. Finally, the intermediate calculation results are rewritten into RAM. This process is repeated until the calculation is complete.

Obviously, since four groups of eight RAMs are used to access the intermediate results, it is necessary to ensure the read and write order of the RAM addresses, realize the orderly access of parameters in each stage, and prevent read and write conflicts.

2) Data Access Control: The parameters of reading and writing in each round of butterfly operations are different, and the address is unique. When reading the parameters, for the convenience of operation, the parameters are rearranged, the RAM is read uniformly, all parameters are obtained and the coefficient pairs are formed. Therefore, when writing the intermediate results, it is necessary to store the output even and odd at different addresses to ensure that the corresponding coefficient pair can be read again from the same address in the next stage.

Algorithm 5 Conversion of the RAM Read Address

Input: N, M , $bstg$, $blop$
Output: RAM read address $raddr$
1: initialize $raddr = 0$
2: $blmt = \log_2 N - \log_2 M - 1$
3: $blso = blmt - (bstg + 1)$
4: **if** $(bstg < blmt)$ **then**
5: **if** $(\neg\ (blop\ \ \&\ \ 0\text{x}01))$ **then**
6: $raddr = (blop >> 1) + ((blop >> (blso+1)) << blso)$
7: **else** $raddr = (1 << blso) + (blop >> 1) + ((blop >> (blso + 1)) << blso)$
8: **end if**
9: **else** $raddr = blop$
10: **end if**
11: return $raddr$

Algorithm 6 Conversion of the RAM Write Address

Input: N, M , $bstg$, $blop$
Output: RAM write address $waddre$, $waddro$
1: initialize $waddre = waddro = 0$
2: $blmt = \log_2 N - \log_2 M - 1$
3: $blso = blmt - (bstg + 1)$
4: **if** $(bstg < blmt)$ **then**
5: $waddre = (blop >> 1) + ((blop >> (blso + 1)) << blso)$
6: $waddro = (blop >> 1) + ((blop >> (blso + 1)) << blso) + (1 << blso)$
7: **else**
8: $waddre = blop$
9: $waddro = blop$
10: **end if**
11: return $waddre$, $waddro$

Specifically, for the NTT operation of N points and M butterfly operation units, the read and write addresses of the RAM are changed according to the stage of the butterfly operation and the number of loops. Among them, the number of loops blop is used to count the butterfly operations in each stage. When the operation starts, it is set to 0, and each clock is incremented by 1. After accumulating to $(N \gg (\log_2 M + 1)) - 1$, the loop waits for the start of the butterfly operation of the next stage and starts counting from 0 again. The butterfly operation stage bstg changes according to the blop count. Every time the current stage calculation is completed, 1 is added to bstg. When bstg accumulates to $\log_2 N + 1$, it is reset to 0. Then, for the NTT operation, the conversion process of the RAM read address is shown in Algorithm 5.

In step 3 of Algorithm 5, when bstg is greater than blmt, the read address of the RAM can be obtained directly according to bstg. Otherwise, additional calculations need to be performed according to the current bstg and blop to obtain the read address of the RAM.

Similarly, the write RAM operation to NTT is shown in Algorithm 6, where waddre and waddro are the write addresses of the two results of even and odd. The reading and writing of INTT RAM is similar to that of NTT and will not be repeated here.

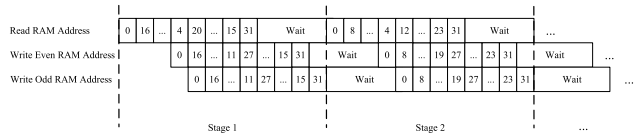

Fig. 9. Schematic of the RAM read and write waiting process.

TABLE II
SHA-3 FUNCTIONS IN THE PQC STANDARDIZATION SCHEME

PQC Scheme	SHA-3 Function
CRYSTAL-Kyber	SHA3-256/512, SHAKE128/256
CRYSTAL-Dilithium	SHAKE128/256
FALCON	SHAKE256

Since the read and write processes of the RAM need to obtain the address first, the data can be read and written, that is, the operation of RAM requires a certain time interval. The butterfly operation is performed in a pipelined way and can process data continuously. Therefore, it is necessary to handle the relationship between the pipeline and RAM to prevent data interruption and errors.

Take NTT with 256 points and four butterfly computing units as an example. As shown in Fig. 9, when data are initially sent to the pipeline, the RAM read/write address is calculated in advance, and the address is tapped and cached. Specifically, the read address of the RAM is sent first, and the write address of the RAM is sent after an interval of eight clocks. In this way, reading first and then writing not only ensures the continuous reading of data but also continuously writes data after the calculation is completed and prevents RAM address conflicts. The waiting time for each stage is 11 clock cycles.

D. Scalable optimization of SHA-3

1) SHA-3 Algorithm Analysis: Among the many LBC schemes, SHA-3 can not only hash key information, but also generate a random number bit stream [52], which can be used in the sampling module to generate sampling values that meet the conditions. As an important core operation, SHA-3 accounts for more than 20% of resource overhead and processing cycles [53]. As shown in Table II, the use of the SHA-3 function in the NIST PQC standardization scheme is given, and both SHA3-256/512 and SHAKE128/256 are used. Therefore, it is particularly important to optimize SHA-3 at high speeds and high efficiency on the hardware, realize the scalable SHA-3 algorithm, and achieve a compromise between performance and space.

The Keccak algorithm is an algorithm selected by the SHA-3 standard. It is implemented based on the sponge structure and requires 24 rounds of iterative operations. The Keccak algorithm first performs special padding on the input data, then performs five-step iterative permutation operations on the 3-D matrix, and finally outputs the hash value. The permutation function Keccak-f[1600] takes a 1600-bit state, converts it into a 3-D array of $5 \times 5 \times 64$ as input, and performs 24 rounds of transformations based on the array. Each round of transformations includes θ, ρ, π, χ, and τ 5 steps. Specifically, the operation of each calculation step is as follows.

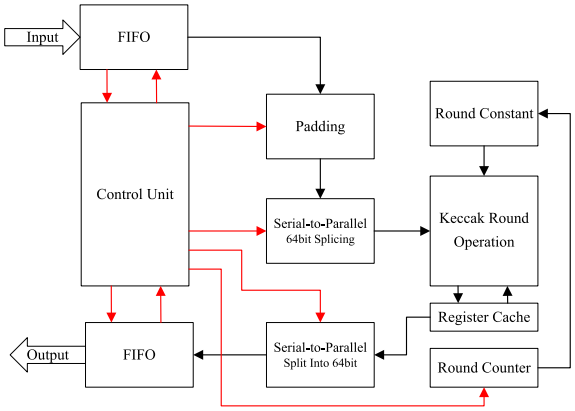

Fig. 10. SHA-3 overall structure.

1) *Step θ ($0 \le x, y < 5, 0 \le z < 64$):* $C[x, y, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$; $D[x, z] = C[x-1, z] \oplus \mathrm{ROT}(C[x+1, z], 1)$; $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$.
2) *Step ρ ($0 \le x, y < 5$):* $A[x, y, z] = \mathrm{ROT}(A'[x, y, z], r[x, y])$.
3) *Step π ($0 \le x, y < 5, 0 \le z < 64$):* $B[y, 2x + 3y, z] = A[x, y, z]$.
4) *Step χ ($0 \le x, y < 5, 0 \le z < 64$):* $A'[x, y, z] = B[x, y, z] \oplus (\neg(B[x + 1, y, z]) \wedge (B[x + 2, y, z]))$.
5) *Step τ ($0 \le i < 24$):* $A'[0, 0, z] = A'[0, 0, z] \oplus RC[i]$.

In the above operation steps, x and y represent operations within the range of modulo 5, and z is modulo 64. $A[x, y, z]$ is the state array, and $B[x, y, z]$, $C[x, y, z]$, and $D[x, z]$ are the intermediate results. $\oplus$ stands for the bitwise XOR operation, $\neg$ denotes the NOT operation, $\wedge$ denotes the AND operation, and the ROT function stands for a circular shift. $r[x][y]$ and $RC[i]$ are the number of shifted bits and the round constant, respectively.

2) SHA-3 Structure Optimization: For the hardware optimization of SHA-3, first, merge processing is adopted, and the five steps of θ, ρ, π, χ, and τ are merged into one round so that one iteration operation is performed in one clock cycle. The entire Keccak operation only needs 24 cycles, improving operation efficiency. Second, for the I/O data interface, the 64 bits width is used as the standard for unified input and output, and when executing Keccak-f[1600], the data are spliced into 1600 bits and then split into 64 bits after the calculation is completed to reduce the resource overhead and improve throughput. Again, for SHA3-256/512 and SHAKE128/256, the core operation Keccak-f[1600] is the same, and only message splicing and result truncation are different. Then, the entire SHA-3 hardware structure can be mainly divided into a control unit, a padding unit, a round operation unit, and a serial-to-parallel conversion unit. Finally, FIFO is used as a cache, the data to be calculated are prestored in FIFO, and the result is also stored in FIFO, reducing the intermediate waiting time. The overall structure of SHA-3 is shown in Fig. 10.

In Fig. 10, the functions of each module are as follows.

1) *Control Module:* schedule and control the overall execution of SHA-3, including the input and output, padding, serial-to-parallel conversion, and round calculation.

2) *Padding Module:* pad the message according to the input calculation mode. When calculating SHA3-256/512, pad with 0x06 and 0x80; when calculating SHAKE128/256, pad with 0x1f and 0x80.
3) *Serial-to-Parallel Conversion Module:* assemble 64-bit data into 1600 bits as the input of Keccak or split the operation result into 64 bits for output.
4) *Round Operation Module:* execute core function, one clock completes one round operation.
5) *Round Counter Module:* after the operation starts, the round counter adds 1 every clock cycle, and the value ranges from 0 to 23.
6) *Round Constant Module:* stores the constant of each round and outputs the constant of the corresponding round according to the round counter.

E. Scalable Design

Multiple polynomial multiplications are involved in the LBC algorithm. Taking the CRYSTAL-Dilithium algorithm [54] as an example, in the key generation stage, it is necessary to generate a polynomial matrix $\mathbf{A}$ and polynomial vectors $\mathbf{s_1}$ and $\mathbf{s_2}$ and calculate $\mathbf{t} = \mathbf{As_1} + \mathbf{s_2}$. In the signature stage, it is necessary to calculate $\mathbf{w} = \mathbf{Ay}$ and $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ multiple times, and in the verification stage, it is necessary to calculate $\mathbf{Az} - c\mathbf{t_1}$. Here, $\mathbf{t}$, $\mathbf{w}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{t_1}$ are polynomial vectors, and c is a polynomial. Obviously, each polynomial operation needs to perform a large number of NTT and INTT operations. To further accelerate the calculation of the LBC algorithm, multiple NTT and SHA-3 modules are instantiated on the FPGA. Each module is independent of each other and managed by finite-state machine (FSM) scheduling to realize the parallel processing of polynomials and improve scalability.

1) Parallelization of NTT: To improve the polynomial operation speed in LBC, multiple NTT modules are instantiated in parallel, and each module is connected with a loosely coupled architecture, as shown in Fig. 11. Under the premise of a unified interface, the external data are input into the corresponding RAM with FIFO through the initialization operation. Subsequently, the control module calls the corresponding FSM to read data from the RAM according to the currently executed operations, such as NTT, INTT, and PWM, and sends them to the corresponding NTT module for calculation. Second, the intermediate results calculated by the NTT module are rewritten into RAM to achieve resource reuse. Finally, after the entire calculation is completed, the control module reads data from the corresponding RAM and sends it to the output FIFO for output.

For example, in the CRYSTAL-Dilithium signature stage, NTT1 and NTT2 modules can calculate $\mathbf{w} = \mathbf{Ay}$ and $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ in parallel, shortening the signature time. Obviously, through this loosely coupled architecture, the calculation of multilattice ciphers can be adapted, and it has good scalability.

2) SHA-3 Parallelization: To accelerate the generation of sampling data in LBC, multiple SHA-3 modules are used for parallel calculation. Taking CRYSTAL-Dilithium as an example, multiple discrete Gaussian sampling and binomial distribution sampling operations need to be performed in the key generation, signature, and verification stages to generate polynomial matrix $\mathbf{A}$; polynomial vectors $\mathbf{s_1}$, $\mathbf{s_2}$, and $\mathbf{y}$; and polynomial c. To speed up the generation of $\mathbf{A}$, two SHA-3 parallel calculations are used, and the random coefficients of sampling are judged in parallel. For the generation of $\mathbf{s_1}$, $\mathbf{s_2}$, $\mathbf{y}$, and c, a shared SHA-3 algorithm is used for calculation, followed by serial-to-parallel conversion and parallel sampling. The corresponding structure of SHA-3 parallelization is shown in Fig. 12.

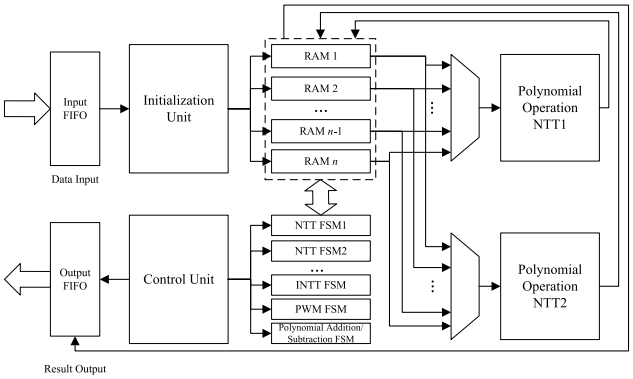

Fig. 11. NTT operation parallel structure.

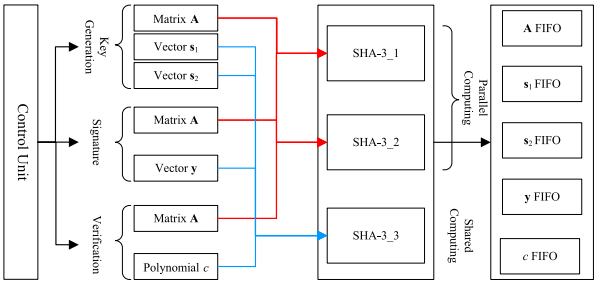

Fig. 12. Multi-SHA-3 parallel computing structure.

IV. Experimental Results and Analysis

A. Experimental Environment

The hardware platform of the experiment in this article is an FPGA accelerator. The chip type is xcku060-ffva1156-2-i of Xilinx. Its lookup table (LUT) resource is 331 680, Flip-flop (FF) register resource is 663 360, BRAM storage resource is 1080, and DSP computing resource is 2760. The software platform is Vivado 2019.2, which integrates the design, simulation, synthesis, routing, and implementation.

B. Modulus Reduction Comparison

There are various algorithms for modulus reduction, such as Montgomery [13], Barrett [15], and K^2-RED. When calculating $z = xy \bmod q$, the result of the Montgomery algorithm is $zR^{-1} \bmod q$, where R is 2^n and R^{-1} is the inverse of R modulo q. Then, there is $zR^{-1} \bmod q = ((z + q \times ((z \times q') \bmod R))/R) \bmod q$, where $q' = -q^{-1} \bmod R$. Therefore, let $W = \log_2(2n)$ and redefine $R = 2^W$, then divide the reduction operation into W-sized parts instead of performing it all at once, and the result can be obtained only by iterating $L = \lceil K/W \rceil$ times, where K is the bit width of q. For the Barrett algorithm, if there exists s such that $p = xy = s \times q + z$, then

TABLE III
COMPARISON OF THREE MODULAR REDUCTION ALGORITHMS

Modular Reduction	Operation Explain	Mode	Scalability	LUTs	FFs	CARRY8s
Montgomery [13]	2 shifts, 7 additions, 2 multiplications	1-stage pipeline	✓	80	13	9
Barrett [15]	3 shifts, 9 additions, 1 MUX	1-stage pipeline	x	37	2	8
K^2-RED	4 shifts, 9 additions	1-stage pipeline	✓	72	0	6

TABLE IV
OVERALL IMPLEMENTATION OF NTT

Module	Mode	LUTs	FFs	BRAMs	DSPs
Butterfly unit	pipeline	615	604	0	3
Control unit	serial	580	384	0	0
Multi-RAM channels	parallel	112	0	1.5	0
Overall of NTT	serial parallel hybrid	1472	1324	1.5	3

the result z can be obtained. The Barrett algorithm calculates the approximate value s' of s. Let $\mu = \lfloor 2^{2n}/q \rfloor$, $s' = \lfloor p/q \rfloor$, then $s' = \lfloor p/q \rfloor \approx \lfloor p \times \mu/2^{2n} \rfloor \approx \lfloor \lfloor p/2^n \rfloor \times \mu/2^n \rfloor$, where $p/2^n$ and $\mu/2^n$ can be realized by shifting to the right, and $z = p - s' \times q$ is finally calculated. As shown in Table III, which gives the realization of three algorithms with modulus $q = 3329$.

In Table III, the three algorithms are implemented in a pipelined manner. Among them, the Montgomery algorithm takes up the most resources because it requires two multiplication operations. The Barrett algorithm takes up the least amount of resources because it is optimized and stores intermediate parameters in a precomputed manner, so multiplexed selection is needed. The K^2-RED algorithm occupies moderate resources and consumes the least adder CARRY8 because the algorithm simplifies part of the operation process by shifting. In terms of scalability, since the Barrett algorithm needs to be optimized for a specific modulus, once the modulus changes, it still needs to be optimized again, and the scalability is poor. However, the Montgomery algorithm and the K^2-RED algorithm can adapt to different moduli only by adjusting some configuration parameters and have good scalability.

C. NTT Implementation

Taking the modulus $q = 77\,681$ and the polynomial size $n = 256$ as an example, the overall implementation of NTT is given. Its operating frequency is 285 MHz, and the specific resource occupation is shown in Table IV.

It can be seen from Table IV that the resource occupation of the butterfly operation unit is relatively balanced and only consumes three DSPs to complete the modular multiplication operation. The control unit consumes more LUT resources because it needs to calculate various counters, addresses, and control enable signals, including the butterfly operation round, the number of iterations in this round, the read and write addresses of RAM, and the read and write enable signal and wait control. Second, for multiple RAM channels, all parameters in the NTT/INTT/PWM calculation process need to be stored, and only part of the FPGA block BRAM resources are consumed. Finally, for the overall implementation of NTT, the resource occupation ratio of LUT is 0.53%, FF is 0.20%, BRAM is 0.14%, DSP is 0.11%, and the resource consumption is less.

TABLE V
REALIZATION OF NTT OF VARIOUS POLYNOMIALS

Modulus q	$(n, \lceil \log_2 q \rceil)$	LUTs	FFs	BRAMs	DSPs	Frequency / MHz
3329	(256, 12)	1914	2249	3	3	275
7681	(256, 13)	1472	1324	1.5	3	285
12289	(1024, 14)	1644	1705	3	3	250
8380417	(256, 23)	1623	1325	1.5	3	255
206158430209	(512, 38)	2021	1794	3	9	180
206158430209	(2048, 38)	2152	1968	7.5	9	180

In addition, the NTT implementations of other moduli q and polynomials of different sizes are shown in Table V.

It can be seen from Table V that the larger the bit width of modulus q involved in the operation is, the larger the degree n of the polynomial, the more resources it occupies, and the operating frequency will decrease accordingly. However, the NTT implementation of various polynomials occupies relatively moderate resources, and the clock frequency of the commonly used modulus q is above 200 MHz, which can meet the needs of high-performance computing of LBC.

D. Performance Analysis

For the parallel situation of multiple butterfly units, the realization of the Dilithium algorithm modulus $q = 8\,380\,417$ is given in Table VI.

It can be seen from Table VI that as the number of butterfly units increases, increasingly more resources are consumed, and the clock frequency decreases. When 32 butterfly units work in parallel, the clock is only 135 MHz. However, the NTT, INTT, and PWM calculation cycles are also continuously reduced, and the calculation speed is improved. In addition, when 16 butterfly units work in parallel, the performance is the best, and when 32 butterfly units work in parallel, the performance decreases. This is mainly determined by the clock frequency and the calculation period. If the resource consumption of the 32 butterfly units can be further reduced and the working clock frequency can be increased, its performance should be improved.

E. Comparison With Other Schemes

As shown in Table VII, the FPGA implementation of other schemes is comprehensively compared in terms of scalability, NTT clock cycles, frequency, resource consumption, and time.

It can be seen from Table VII that when $n = 1024$ and $q = 3329$, schemes [11], [15], and [51] have been specially optimized and are superior to our scheme in terms of resource consumption and AT product. Especially in the scheme [11], its NTT cycle is the shortest, which is 69. And the scheme [51] has the smallest AT product. However, none of these methods

TABLE VI
NTT MULTIBUTTERFLY UNIT PARALLEL HARDWARE INDICATORS

Number of Butterfly Units	LUTs	FFs	BRAMs	DSPs	Frequency / MHz	Clock Cycle (NTT/INTT/PWM)	Performance / p/s (NTT/INTT/PWM)
1	1623	1325	1.5	3	255	1052/1052/3684	242395/242395/69218
4	5478	4955	6	12	250	284/284/998	880282/880282/250501
8	11006	8791	12	24	220	156/156/550	1410256/1410256/400000
16	26520	15171	24	48	194	113/113/385	1716814/1716814/541899
32	63533	27488	48	96	135	95/95/319	1421053/1421053/423197

TABLE VII
COMPREHENSIVE COMPARISON WITH OTHER FPGA SCHEMES

Scheme	Device	Scal-ability	NTT Cycle	Frequency / MHz	LUTs/AT	FFs/AT	BRAMs/AT	DSPs/AT	Time / μ s
$n = 256$, $q = 3329$, 12 bits									
Yaman et al. [11]	Artix-7	x	69	172	9508/3814.26	2684/1076.72	35/14.04	16/6.42	0.4
Xing et al. [15]	Artix-7	x	512	161	1737/5523.88	1167/3711.20	3/9.54	2/6.36	3.18
Deryaet al. [38]	Virtex-7	✓	84	167	61000/30682.63	17000/8550.90	48/24.14	256/128.77	0.50
Bisheh-Niasar et al. [51]	Artix-7	x	324	222	801/1169.03	717/1046.43	4/5.84	2/2.92	1.46
Ours	XCKU060	✓	100	200	24186/12093.00	14756/7378.00	24/12.00	48/24.00	0.50
$n = 1024$, $q = 12289$, 14 bits									
Mert et al. [13]	Virtex-7	✓	200	125	17188/27500.80	–	48/76.80	96/153.60	1.6
Chen et al. [36]	Virtex-7	x	654	227	6300/18150.66	2124/6119.37	10/28.81	27/77.79	2.88
Zhang et al. [48]	XC7Z020	x	2569	244	847/8917.80	375/3948.26	6/63.17	2/21.06	10.53
Bisheh-Niasar et al. [51]	Artix-7	x	1591	234	798/5425.72	715/4861.39	2/13.60	4/27.20	6.8
Ours	XCKU060	✓	343	200	22648/38841.32	15030/25776.45	24/41.16	48/82.32	1.72
$n = 256$, $q = 8380417$, 23 bits									
Land et al. [9]	XC7A100T	x	533	311	524/898.05	759/1300.79	1/1.71	17/29.14	1.71
Zhou et al. [19]	Artix-7	x	1170	216	2044/11071.67	–	6/32.50	12/65.00	5.42
Deryaet al. [38]	Artix-7	✓	103	126	63000/51500.00	18000/14714.29	48/39.24	256/209.27	0.82
Ours	XCKU060	✓	113	194	26520/15447.22	15171/8836.72	24/13.98	48/27.96	0.58
$n = 1024$, $q = 3221225473$, 32 bits									
Hu et al. [23]	Virtex-7	x	200	161	44000/52800.00	2600/3120.00	128/153.60	192/230.40	1.20
Su et al. [55]	Virtex-7	✓	–	250	10272/26707.20	6704/17430.40	79/205.40	80/208.00	2.60
Ours	XCKU060	✓	343	190	25609/46230.98	16502/29790.45	28.5/51.45	48/86.65	1.81

AT = Area $\times$ Time

are scalable. The scheme [38] has good scalability, but its resource consumption is too high, resulting in a high AT product, except that the NTT cycle is better than our scheme, and other aspects are poor.

When $n = 1024$ and $q = 12\,289$, the scheme [13] has the lowest NTT cycle while ensuring scalability and the AT product of its LUT is better than our scheme. However, it consumes more DSP and RAM, resulting in a higher AT product for them. The scheme [36] optimizes NTT based on radix-4, which reduces resource consumption and the NTT cycle and has a better AT product. However, it only implements NTT and INTT operations and does not support PWM operations. The scheme [48] is specially optimized for $q = 12\,289$. Although it has a better AT product, it has the longest NTT cycle and poor scalability. The scheme [51] instantiates four butterfly units with a better AT product, but the NTT cycle is longer and the scalability is poor.

When $n = 256$ and $q = 8\,380\,417$, the scheme [9] is specifically optimized and only instantiates two butterfly units, occupying the least resources and having the lowest AT product. However, this scheme is not scalable and has poor flexibility. The scheme [19] implements the NTT algorithm by combining software and hardware on an FPGA. But its NTT cycle is longer, the time is much higher than our scheme, and the RAM and DSP AT products are higher. The scheme [38] has better scalability, but this scheme does not support other moduli exceeding 32 bits. Compared with our scheme, it consumes more resources, and all AT products are higher.

When $n = 1024$ and $q = 3\,221\,225\,473$, the scheme [23] only implements the NTT operation, which has poor scalability and consumes more resources, and has a higher AT product for all aspects except FF. The scheme [55] has a higher frequency and consumes less LUT and FF, and the AT product of LUT and FF is better than our scheme. However, this scheme uses more RAM and DSP and has a longer computation time, resulting in a higher AT product of RAM and DSP.

In summary, compared to most of the schemes, our scheme is scalable and can adapt to polynomial multiplication operations of various bit widths and moduli. In terms of the NTT clock cycle, this article has a shorter calculation period than most other schemes and ensures a higher operating frequency. In terms of resource consumption, our scheme consumes more resources than most other schemes. This is

TABLE VIII
RESOURCE OCCUPATION AND CALCULATION CYCLE OF EACH MODULE IN THE DILITHIUM ALGORITHM

Module	LUTs	FFs	BRAMs	DSPs	Frequency / MHz	Clock Cycle
Key generation	5318	4635	19.5	12	250	5296
Signature	13474	11438	51	24	250	18521
Verification	7245	5605	27	12	250	7980
Sampling	20918	15868	0	0	235	1859
Dilithium	47013	37650	97.5	48	225	33656

because this article uses multiple butterfly units in parallel and supports NTT, INTT, and PWM operations, and to adapt various parameters, it needs to consume more resources. Other schemes are specifically optimized for fixed parameters, and some schemes only implement NTT calculations and consume fewer resources.

F. Application Analysis

Dilithium is a standard postquantum digital signature algorithm that includes the main steps of key generation, signature, verification, and sampling. To verify the effectiveness of the scheme in this article, the NTT module is composed of four butterfly units in parallel, the Dilithium algorithm is composed of four NTT modules and three SHA-3 modules in parallel, and the entire calculation process is completed through FSM control. In Table VIII, the resource occupation and calculation cycle of each module in the Dilithium algorithm are given.

As seen from Table VIII, the overall resource occupation of the Dilithium algorithm is moderate, the operating frequency is 225 MHz, and the clock numbers of the key generation, signature, and verification stages are 5.3, 18.5, and 8.0 k, respectively, which can be completed in a relatively short time with high performance. If a single NTT module is used, key generation, signature, and verification can only call one NTT module serially, and cannot work in parallel. At this point, it takes a total of 31 797 clock cycles to execute all operations at once, which is inefficient. Similarly, if a single SHA-3 module is used, the generation of polynomial matrix $\mathbf{A}$ (1859×2 clock cycles) and polynomial vectors $\mathbf{s_1}$ and $\mathbf{s_2}$ (total 1378 clock cycles) in the sampling stage can only be executed serially, and 5096 clock cycles are required. Furthermore, key generation, signature, and verification can be executed independently and used to communicate with the FIFO interface, which has good scalability.

For the SHA-3 algorithm in Dilithium, we compare it with other schemes as shown in Table IX. In Table IX, the SHA-3 algorithm for all schemes is 24 clock cycles. The scheme [17] implements the two algorithms of SHAKE128 and SHAKE256, respectively, which have a higher frequency but no scalability. The scheme [56] has scalability and takes up fewer resources, but its frequency is only 256 MHz. The scheme [57] has scalability but takes up more resources. Compared with other schemes, ours occupies moderate resources, operates at 405 MHz, and has scalability, which balances the resources and performance.

Furthermore, a comprehensive comparison was made with other Dilithium schemes, as shown in Table X.

TABLE IX
COMPARISON WITH OTHER SHA-3 SCHEMES

Scheme	LUTs	FFs	Frequency / MHz	Clock Cycle	Scalability
Ricci et al. [17]	6736	3216	587.2	24	x
Beckwith et al. [56]	5483	4451	256	24	✓
Mert et al. [57]	8738	3482	260	24	✓
Ours	5962	4490	405	24	✓

In Table X, the schemes in [9] and [57] have consumed less resources; however, the key generation, signature, and verification clock cycles are long, and the performance is poor. For the scheme in [17], key generation, signature, and verification can be performed independently, and the operating frequencies are 350, 333, and 158 MHz, but it obviously consumes more resources and the calculation cycle is longer. The scheme in [56] has a higher frequency and a lower clock cycle number, but it is implemented in a shared NTT architecture; the key generation, signature, and verification cannot be parallelized; and the scalability is slightly worse. The scheme [58] consumes moderate resources; however, the signature calculation cycle is long, and the operating frequency is low, so the overall performance is poor. Obviously, the scheme proposed in this article has a relatively balanced overall resource consumption, high operating frequency, and low computing cycle and gives full play to the computing advantages of an FPGA. Moreover, key generation, signature, and verification can be executed in parallel, which can further shorten the overall calculation time in practical applications, demonstrating the scheme's good application value.

G. Discussions

This article implements scalable and parallel optimization of the NTT on an FPGA, which can be parallelized with an appropriate number of butterfly units for resource-constrained, high-throughput, and high-performance scenarios. This scheme is more universal and efficient than other methods.

In terms of antiside channel attack, in the calculation process of NTT, the input data is unknown, and the corresponding results are unpredictable, but the twiddle factor is the only known data, which remains unchanged throughout the calculation process. Therefore, the constant twiddle factors can be masked using the properties of $\omega^n \equiv 1(\mathrm{mod}q)$ and randomization techniques to resist differential power analysis (DPA). Second, the Chinese remainder theorem (CRT) can be used in the LBC algorithm to resist differential fault analysis (DFA). We can find a modulus q' that is relatively prime to the modulus q, combine it to form a new modulus $\widehat{q} = q \times q'$, and then find a polynomial function g to generate check values. Then, combine the coefficients of the ring modulo q with the constant coefficients of the ring modulo q' on the ring modulo $\widehat{q}$ to perform operations. After a sequence of computations is finished, new check values are obtained in the final result by the ring modulo q'. And comparing them with the predetermined check values, it can be found whether the PQC algorithm has been attacked by DPA. Finally, we can further increase the attack threshold by configuring different

TABLE X
COMPREHENSIVE COMPARISON WITH OTHER DILITHIUM SCHEMES

Scheme	Device	LUTs	FFs	BRAMs	DSPs	Frequency / MHz	Key Generation	Signature	Verification
Land et al. [9]	Artix-7 XC7A100T	27433	10681	15	45	163	18761	76613	19687
Ricci et al. [17]	Virtex 7 Ultrascale+	184382	146494	178	1463	350/333/158	12600	18338	10546
Beckwith et al. [56]	Virtex UltraScale+	53907	28435	29	16	256	4875	10945	6582
Mert et al. [57]	Zynq UltraScale+ ZCU102	19100	9300	24	4	200	14183	30358	15044
Zhao et al. [58]	Artix-7 XC7Z020	29998	10366	11	10	96.9	4172	28091	4422
Ours	XCKU060	47013	37650	97.5	48	225	5296	18521	7980

numbers of butterfly units and randomizing the RAM read and write addresses with a shuffling countermeasure.

In terms of hardware platform implementation, FPGA development is very difficult and has high requirements for algorithm structure, chip resources, layout, and routing and a unified architecture needs to be provided. When facing NTT calculations with different moduli, the reconfigurable feature of an FPGA can be fully utilized to provide flexible support. Although ASIC has the highest computing performance, once the algorithm is determined, its structure cannot be changed, and it cannot adapt to the calculation of NTT of various moduli. Additionally, an ASIC needs to be developed using an FPGA as a prototype. Therefore, an FPGA also provides a theoretical basis and guiding significance for the ASIC PQC chip. For system on chip (SoC), it is realized by ARM+FPGA, software, and hardware cooperation. Among them, the FPGA is used to accelerate NTT operation, and ARM is used for software control. However, limited by the communication bandwidth between ARM and FPGA, it is not suitable for high-throughput scenarios. The FPGA generally has fewer resources, which limits the frequency and performance of the algorithm. Therefore, SoC can be applied to the Internet of Things (IoT) and edge computing.

V. CONCLUSION

LBC schemes will play a vital role in future PQC standards and even in the security of future information systems. This article analyzes the NTT, INTT, and PWM algorithms in depth through the description of the LBC algorithm. Then, pipeline technology is used to realize the optimization of the butterfly operation unit, and multichannel RAM is used to optimize parameter access and reduce the overall calculation delay. Second, the K^2-RED module reduction algorithm and the SHA-3 algorithm are optimized to improve the scalability of the algorithm to adapt to various parameters. Finally, the overall architecture of an FPGA is realized by using a parallel butterfly operation unit, an NTT parallel unit, and an SHA3-parallel unit, which improves the calculation efficiency. The experimental analysis and results show that the implementation scheme in this article has high research and practical application value.

NTT with a larger radix (e.g., radix-4) may achieve faster speed performance. Therefore, our future research will focus on using a larger radix to design NTT and accelerate the algorithm with various levels of parallelism while ensuring higher computing performance and scalability.

In practical applications, side-channel analysis attacks are still the main types of cryptographic attacks. Therefore, providing a security protection strategy for LBC algorithms at the hardware level is very important. This not only increases additional resource consumption, but also increases the difficulty of the design, which still needs to be explored and solved.

REFERENCES

[1] V. Chamola, A. Jolfaei, V. Chanana, P. Parashari, and V. Hassija, "Information security in the post quantum era for 5G and beyond networks: Threats to existing cryptography, and post-quantum cryptography," *Comput. Commun.*, vol. 176, pp. 99–118, Aug. 2021.

[2] J. Howe, T. Prest, and D. Apon, "SoK: How (not) to design and implement post-quantum cryptography," in *Topics in Cryptology—CT-RSA 2021*. Cham, Switzerland: Springer, May 2021, pp. 444–477.

[3] M. Anastasova, R. Azarderakhsh, and M. M. Kermani, "Fast strategies for the implementation of SIKE round 3 on ARM cortex-M4," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 10, pp. 4129–4141, Oct. 2021.

[4] A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao, "Supersingular isogeny Diffie–Hellman key exchange on 64-bit ARM," *IEEE Trans. Depend. Sec. Comput.*, vol. 16, no. 5, pp. 902–912, Sep. 2019.

[5] A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao, "Towards optimized and constant-time CSIDH on embedded devices," in *Constructive Side-Channel Analysis and Secure Design*. Cham, Switzerland: Springer, Apr. 2019, pp. 215–231.

[6] S. Bayat-Sarmadi, M. Mozaffari Kermani, R. Azarderakhsh, and C.-Y. Lee, "Dual-basis superserial multipliers for secure applications and lightweight cryptographic architectures," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 2, pp. 125–129, Feb. 2014.

[7] T. Fritzmann et al., "Masked accelerators and instruction set extensions for post-quantum cryptography," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, no. 1, pp. 414–460, Nov. 2021.

[8] W. Guo and S. Li, "Area-efficient modular reduction structure and memory access scheme for NTT," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–5.

[9] G. Land, P. Sasdrich, and T. Güneysu, "A hard crystal-implementing dilithium on reconfigurable hardware," in *Smart Card Research and Advanced Applications*. Cham, Switzerland: Springer, Mar. 2022, pp. 210–230.

[10] T. T. Nguyen, S. Kim, Y. Eom, and H. Lee, "Area-time efficient hardware architecture for CRYSTALS-kyber," *Appl. Sci.*, vol. 12, no. 11, p. 5305, May 2022.

[11] F. Yaman, 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 Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 1020–1025.

[12] Y. Huang, M. Huang, Z. Lei, and J. Wu, "A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse," *IEICE Electron. Exp.*, vol. 17, no. 17, 2020, Art. no. 20200234.

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

[14] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture," in *Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2019, pp. 253–260.

[15] Y. Xing and S. Li, "A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 2, pp. 328–356, Feb. 2021.

[16] S. Ricci, P. Jedlicka, P. Cíbik, P. Dzurenda, L. Malina, and J. Hajny, "Towards CRYSTALS-Kyber VHDL implementation," in *Proc. 18th Int. Conf. Secur. Cryptogr. (SECRYPT)*. Setúbal, Portugal: SciTePress, 2021, pp. 760–765.

[17] S. Ricci et al., "Implementing CRYSTALS-dilithium signature scheme on FPGAs," in *Proc. 16th Int. Conf. Availability, Rel. Secur.*, Aug. 2021, pp. 1–11.

[18] L. Ma, X. Wu, and G. Bai, "Parallel polynomial multiplication optimized scheme for CRYSTALS-KYBER post-quantum cryptosystem based on FPGA," in *Proc. Int. Conf. Commun., Inf. Syst. Comput. Eng. (CISCE)*, May 2021, pp. 361–365.

[19] Z. Zhou, D. He, Z. Liu, M. Luo, and K.-K.-R. Choo, "A software/hardware co-design of crystals-dilithium signature scheme," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, no. 2, pp. 1–21, Jun. 2021.

[20] R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "High-performance FPGA accelerator for SIKE," *IEEE Trans. Comput.*, vol. 71, no. 6, pp. 1237–1248, Jun. 2022.

[21] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of CRYSTALS-dilithium," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2021, pp. 1–10.

[22] D.-e.-S. Kundi, Y. Zhang, C. Wang, A. Khalid, M. O'Neill, and W. Liu, "Ultra high-speed polynomial multiplications for lattice-based cryptography on FPGAs," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 4, pp. 1993–2005, Oct. 2022.

[23] X. Hu, M. Li, J. Tian, and Z. Wang, "DARM: A low-complexity and fast modular multiplier for lattice-based cryptography," in *Proc. IEEE 32nd Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Jul. 2021, pp. 175–178.

[24] R. Azarderakhsh, B. Koziel, S. F. Langroudi, and M. M. Kermani, "FPGA-SIDH: High-performance implementation of supersingular isogeny Diffie–Hellman key-exchange protocol on FPGA," *Proc. ePrint*, vol. 672, pp. 1–18, Jul. 2016.

[25] Z. Chen, Y. Ma, T. Chen, J. Lin, and J. Jing, "Towards efficient kyber on FPGAs: A processor for vector of polynomials," in *Proc. 25th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2020, pp. 247–252.

[26] D. Li, A. Pakala, and K. Yang, "MeNTT: A compact and efficient processing-in-memory number theoretic transform (NTT) accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 5, pp. 579–588, May 2022.

[27] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "CryptoPIM: In-memory acceleration for lattice-based cryptographic hardware," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[28] G. Seiler, "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography," *Cryptol. ePrint Arch.*, vol. 39, pp. 1–14, Jan. 2018. [Online]. Available: https://eprint.iacr.org/2018/039

[29] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 4, pp. 239–280, Aug. 2020.

[30] C.-M.-M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on cortex-M4 and AVX2," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 2, pp. 159–188, Feb. 2021.

[31] J. W. Bos, M. Gourjon, J. Renes, T. Schneider, and C. Van Vredendaal, "Masking kyber: First- and higher-order implementations," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 4, pp. 173–214, Aug. 2021.

[32] R. Agrawal, L. Bu, A. Ehret, and M. Kinsy, "Open-source FPGA implementation of post-quantum cryptographic hardware primitives," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 211–217.

[33] T. Zijlstra, K. Bigou, and A. Tisserand, "Lattice-based cryptosystems on FPGA: Parallelization and comparison using HLS," *IEEE Trans. Comput.*, vol. 71, no. 8, pp. 1916–1927, Aug. 2022.

[34] K. Basu, D. Soni, M. Nabeel, and R. Karri, "NIST post-quantum cryptography—A hardware evaluation study," *Cryptol. ePrint Arch.*, vol. 47, pp. 1–16, May 2019. [Online]. Available: https://eprint.iacr.org/2019/047

[35] E. Haleplidis, T. Tsakoulis, A. El-Kady, C. Dimopoulos, O. Koufopavlou, and A. P. Fournaris, "Studying OpenCL-based number theoretic transform for heterogeneous platforms," in *Proc. 24th Euromicro Conf. Digit. Syst. Design (DSD)*, Sep. 2021, pp. 339–346.

[36] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, "CFNTT: Scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, no. 1, pp. 94–126, Nov. 2021.

[37] P. Duong-Ngoc and H. Lee, "Configurable mixed-radix number theoretic transform architecture for lattice-based cryptography," *IEEE Access*, vol. 10, pp. 12732–12741, 2022.

[38] K. Derya, A. C. Mert, E. Öztürk, and E. Savaş, "CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication," *Microprocessors Microsyst.*, vol. 89, Mar. 2022, Art. no. 104451.

[39] 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.

[40] M. Mozaffari-Kermani and A. Reyhani-Masoleh, "Fault detection structures of the S-boxes and the inverse S-boxes for the advanced encryption standard," *J. Electron. Test.*, vol. 25, nos. 4–5, pp. 225–245, Aug. 2009.

[41] S. Ali, X. Guo, R. Karri, and D. Mukhopadhyay, "Fault attacks on AES and their countermeasures," *Secure System Design and Trustable Computing*. Springer, ChamCham, Switzerland: Springer, 2016, pp. 163–208.

[42] B. Koziel, R. Azarderakhsh, and M. M. Kermani, "Low-resource and fast binary Edwards curves cryptography," in *Progress in Cryptology—INDOCRYPT 2015*. Bengaluru, India: Springer, Dec. 2015, pp. 347–369.

[43] E. Dubrova, K. Ngo, and J. Gärtner, "Breaking a fifth-order masked implementation of CRYSTALS-Kyber by copy-paste," *Cryptol. ePrint Arch.*, vol. 1713, pp. 1–22, Dec. 2022. [Online]. Available: https://eprint.iacr.org/2022/1713

[44] A. Berzati, A. C. Viera, M. Chartouni, S. Madec, D. Vergnaud, and D. Vigilant, "A practical template attack on CRYSTALS-Dilithium," *Cryptol. ePrint Arch.*, vol. 50, pp. 1–23, Jan. 2023. [Online]. Available: https://eprint.iacr.org/2023/050

[45] M. Mozaffari-Kermani and R. Azarderakhsh, "Reliable hash trees for post-quantum stateless cryptographic hash-based signatures," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFTS)*, Oct. 2015, pp. 103–108.

[46] M. M. Kermani and R. Azarderakhsh, "Reliable architecture-oblivious error detection schemes for secure cryptographic GCM structures," *IEEE Trans. Rel.*, vol. 68, no. 4, pp. 1347–1355, Dec. 2019.

[47] A. Aghaie, M. M. Kermani, and R. Azarderakhsh, "Fault diagnosis schemes for secure lightweight cryptographic block cipher RECTANGLE benchmarked on FPGA," in *Proc. IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2016, pp. 768–771.

[48] 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, Mar. 2020.

[49] S. Khan, K. Javeed, and Y. A. Shah, "High-speed FPGA implementation of full-word Montgomery multiplier for ECC applications," *Microprocessors Microsyst.*, vol. 62, pp. 91–101, Oct. 2018.

[50] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Cryptology and Network Security*. Cham, Switzerland: Springer, Oct. 2016, pp. 124–139.

[51] 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. Arithmetic (ARITH)*, Jun. 2021, pp. 94–101.

[52] S. Zarei, A. R. Shahmirzadi, H. Soleimany, R. Salarifard, and A. Moradi, "Low-latency Keccak at any arbitrary order," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 4, pp. 388–411, Aug. 2021.

[53] S. Sinha Roy and A. Basso, "High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 4, pp. 443–466, Aug. 2020.

[54] S. Bai et al., "CRYSTALS-dilithium: Algorithm specifications and supporting documentation (version 3.1)," NIST Post-Quantum Cryptogr. Standardization, Feb. 2021, pp. 1–38. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf

[55] Y. Su, B.-L. Yang, C. Yang, Z.-P. Yang, and Y.-W. Liu, "A highly unified reconfigurable multicore architecture to speed up NTT/INTT for homomorphic polynomial multiplication," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 8, pp. 993–1006, Aug. 2022.

[56] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of lattice-based digital signatures," *Cryptol. ePrint Arch.*, vol. 217, pp. 1–18, Feb. 2022. [Online]. Available: https://eprint.iacr.org/2022/217

[57] A. Aikata et al., "A unified cryptoprocessor for lattice-based signature and key-exchange," *IEEE Trans. Comput.*, vol. 72, no. 6, pp. 1568–1580, Jun. 2023.

[58] C. Zhao et al., "A compact and high-performance hardware architecture for CRYSTALS-dilithium," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, no. 1, pp. 270–295, Nov. 2021.