Towards Efficient Kyber on FPGAs: A Processor for Vector of Polynomials

Zhaohui Chen*
School of Computer Science and Technology
University of Chinese Academy of Sciences
Beijing, China 100049
chenzhaohui17@mails.ucas.ac.cn

Yuan Ma†
State Key Laboratory of Information Security
Institute of Information Engineering, CAS
Beijing, China 100095
mayuan@iie.ac.cn

Tianyu Chen
State Key Laboratory of Information Security
Institute of Information Engineering, CAS
Beijing, China 100095
chentianyu@iie.ac.cn

Jingqiang Lin
State Key Laboratory of Information Security
Institute of Information Engineering, CAS
Beijing, China 100095
linjingqiang@iie.ac.cn

Jiwu Jing
School of Computer Science and Technology
University of Chinese Academy of Sciences
Beijing, China 100049
jwjing@ucas.ac.cn

***Abstract*—Kyber is a promising candidate in post-quantum cryptography standardization process. In this paper, we propose a targeted optimization strategy and implement a processor for Kyber on FPGAs. By merging the operations, we cut off 29.4% clock cycles for Kyber512 and 33.3% for Kyber1024 compared with the textbook implementations. We utilize Gentlemen-Sande (GS) butterfly to optimize the Number-Theoretic Transform (NTT) implementation. The bottleneck of memory access is broken taking advantage of a dual-column sequential scheme. We further propose a pipeline architecture for better performance. The optimizations help the processor achieve 31684 NTT operations per second using only 477 LUTs, 237 FFs and 1 DSP. Our strategy is at least 3x more efficient than the state-of-the-art module for NTT with a similar security level.**

I. Introduction

Public key cryptography based on large integer factoring and discrete logarithm problem is widely used in digital signature, electronic authentication and TLS/SSL key exchange, etc. Quantum computers would completely break these cryptosystems with Shor's algorithm [1]. To seek for appropriate substitutes, the National Institute of Standards and Technology (NIST) called for post-quantum public-key encryption, key encapsulation mechanism and digital signature schemes in 2017. Interest in lattice-based cryptography has increased due to the quantum-resistant properties and the potential for high-speed implementation with relatively small key and ciphertext size [2], [3].

Regev [4], [5] introduced Learning With Errors (LWE) problem supported by a theoretical proof of security. However, a large parameter matrix $\mathbf{A}$ limits its efficiency. Lyubashevsky [6] et al. proposed Ring-Learning With Errors (Ring-LWE) over polynomial ring $\mathbb{Z}_q[X]/\left(X^n+1\right)$ to avoid the large parameter. Although Ring-LWE is more practical than the standard LWE, its algebraic structure might enable threatening attacks [7]. Module-Learning With Errors (Module-LWE) hardness assumption proposed in [8] provides a trade-off between security and efficiency with a scalable vector of polynomials ($\mathrm{polyvec}$) structure. As a competitive instance, Kyber [9] algorithm fixes the polynomial ring as $\mathbb{Z}_{7681}[X]/\left(X^{256}+1\right)$. Thus, as the most computationally intensive operation, multiplication over k-dimensional $\mathrm{polyvec}$ can be optimized with a linear algorithm named NTT, which can reduce computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n\log n)$ [10], [11].

There has been increased interest in implementing Ring-LWE on FPGAs due to the potential towards high-performance and compact application scenarios [12]–[17]. A relatively new Ring-LWE-based scheme, known as $\mathrm{NewHope}$ [18], together with its variant $\mathrm{NewHope-Simple}$ [19] were implemented on Artix7 FPGAs with targeted optimization in [16], [17]. However, we have not found any detailed optimization for Module-LWE-based schemes on FPGAs so far. Among the existing works, the high-efficiency implementations like [14] initialize several processing elements in parallel thus more arithmetic and memory instances are required. On the other hand, the NTT algorithm and memory access takes a lot of clock cycles for compact processors like [12], [13]. Thus, implementing a both time and area efficient processor is still a hard work.

In this paper, we design and implement an FPGA-based processor for operations over $\mathrm{polyvec}$ with a good trade-off between area and performance. The efficiency of lattice-based schemes makes a significant improvement compared with the previous hardware implementations. Our contributions are as follows:

1) We optimize the NTT algorithm with GS butterfly. The GS butterfly is used both in forward and inverse NTT in order to utilize the internal DSP adders. The optimization reduces a total of 29.4% clock cycles for $\mathrm{Kyber512}$ and 33.3% for $\mathrm{Kyber1024}$ compared with the textbook implementations.
2) We develop dual-column sequential storage and bit-reversed address accessing. These techniques keep the datapath free of bubble and avoid redundant latency

*Also with State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China 100095.
†This author is the corresponding author.

caused by rearrangement so that our processor attains at least 3x faster than LATINCRYPT'17 [16].

3) We implement a pipeline processor making use of the internal registers of DSP and block RAM (BRAM) slices. Although multiplexing NTT, multiply-accumulate (MACC) and multiply-add (MADD) operations on a single processor, it can perform 31684 NTT calculations for 4-dimensional polyvec at 130MHz.

The remainder of the paper is organized as follows: In Section II we provide a brief mathematical background of Kyber and NTT algorithms. Section III introduces our optimizations and hardware architecture on FPGAs. We implement and compare the results with the Ring-LWE-based NTT module in Section IV. Finally, Section V draws the conclusions.

II. Preliminaries

A. *Symbol Definition*

The polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$ is represented as $\mathcal{R}_q$ in which n is the dimension and q is the modulo. For a polynomial $a \in \mathcal{R}_q$, we suggest that $a = \sum_{i=0}^{n-1} a_i x^i, a_i \in \mathbb{Z}_q$. Let ab and $a \circ b$ denote polynomial multiplication and point-wise multiplication both over $\mathcal{R}_q$, respectively. Vectors of polynomials are written as bold lower-case letters like $\mathbf{s}$. Bold upper-case letters like $\mathbf{A}$ are matrices. If not stated, all the polynomial elements in vectors or matrices are over $\mathcal{R}_q$. Vectors will be column-wise by default. For a vector $\mathbf{a}$ (or matrix $\mathbf{A}$), its transpose is $\mathbf{a}^T$ (or $\mathbf{A}^T$). The centered binomial distribution is defined as β_η for some positive integer η [9], a k-dimensional vector of polynomials can be generated according to the distribution β_η^k. Throughout the paper, we will write normal-font NTT whenever we refer to the general technique and use bold-font $\mathbf{NTT}$ whenever we refer to the corresponding function.

B. Kyber *Algorithm and Parameter Sets*

Kyber is a candidate in NIST post-quantum cryptography standardization consisting of key generation, encryption and decryption algorithms. Given $\mathbf{A}$ as a global matrix of polynomials with coefficients sampled from uniform distribution in NTT domain, a simplified version is shown as follows.

- Kyber.$KeyGen(\mathbf{A})$: Choose two polyvec $\mathbf{s}, \mathbf{e}$ from β_η^k and compute $\mathbf{t} = \mathbf{As} + \mathbf{e}$. The public key is $(\mathbf{A}, \mathbf{t})$ and the private key is $\mathbf{s}$.
- Kyber.$Enc(\mathbf{A}, \mathbf{t}, \mathbf{m})$: The message m is first encoded to $\overline{m}$. Sample polyvec $\mathbf{r}, \mathbf{e}_1$ from β_η^k and e_2 from β_η. The ciphertext then consists of polyvec $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$ and polynomial $v = \mathbf{t}^T\mathbf{r} + e_2 + \overline{m}$.
- Kyber.$Dec(\mathbf{s}, \mathbf{u}, v)$: Compute $m' = v - \mathbf{s}^T\mathbf{u}$ and recover the original message m from m' using a decoder.

For Kyber, the most expensive operations are over vector or matrix of polynomials. A typical computational intensive formula is $\mathbf{As}$ as shown in Eq. (1).

$$\mathbf{As} = \begin{bmatrix} \mathbf{A}(0,0)\,\mathbf{s}(0) + \mathbf{A}(0,1)\,\mathbf{s}(1) \\ \mathbf{A}(1,0)\,\mathbf{s}(0) + \mathbf{A}(1,1)\,\mathbf{s}(1) \end{bmatrix} \tag{1}$$

Kyber provides different post-quantum security levels which enables a fair comparison with the NewHope as shown in Table I. In this paper, we focus on Kyber512 with light parameters and Kyber1024 with paranoid parameters. The fixed $\mathcal{R}_q$ brings an additional advantage and therefore the polyvec processor can be easily configured for different security levels.

TABLE I
Parameter comparisons between Kyber and NewHope in NIST standardization.

Algorithm	Parameter (n/k/q)	Quantum Security (bit)	Ciphertext Size (Byte)
Kyber512	256/2/7681	102	800
Kyber1024	256/4/7681	218	1504
NewHope512	512/1/12289	101	1120
NewHope1024	1024/1/12289	233	2208

C. *Multiplication of Polyvec*

Pöppelmann and Güneysu introduced a NTT-based optimization for polynomial multiplication [10]. In $\mathbb{Z}_{7681}[X]/\left(X^{256}+1\right)$, we can find an n-th root of unity $\omega = 3844$ and its modular square root $\psi = 62$ such that $\psi^2 \equiv \omega \bmod q$. Forward NTT is defined as $\mathbf{NTT}_\omega(a)_i = \sum_{j=0}^{n-1} a_j\omega^{ij} \bmod q, i = 0, 1, \ldots, n-1$. In practice, an efficient variant named negacyclic NTT (negNTT) is widely used to accelerate multiplication of polyvec. We calculate a basic polynomial multiplication as $ab = \mathbf{negNTT}_{\omega^{-1}}(\mathbf{negNTT}_\omega(a) \circ \mathbf{negNTT}_\omega(b))$. We denote point-wise multiply polynomial a by $(1, \psi, \cdots, \psi^{n-1})$ as $\mathbf{PwM}_{\psi^i}(a)$ (define $\mathbf{PwM}_{\psi^{-i}}(a)$ similarly). For $i = 0, 1, \ldots, n-1$, Eq. (2) shows the forward negNTT algorithm.

$$\begin{aligned} \mathbf{negNTT}_\omega(a)_i =& \mathbf{NTT}_\omega(\mathbf{PwM}_\psi(a))_i \\ =& \sum_{j=0}^{n-1} \psi^j a_j \omega^{ij} \bmod q \end{aligned} \tag{2}$$

The forward negNTT adds pre-multiplication by exponent of ψ compared with NTT, which transforms the polynomial to NTT domain for point-wise multiplication. The corresponding inverse negNTT is quite similar. It substitutes ω to its inverse $\omega^{-1} \bmod q = 6584$ and replaces the additional pre-multiplication to a post-multiplication by exponent of $\psi^{-1} \bmod q = 1115$. Finally, each coefficient is multiplied by the scalar $n^{-1} \bmod q = 7651$. Inverse negNTT is denoted as Eq. (3).

$$\begin{aligned} \mathbf{negNTT}_{\omega^{-1}}(a)_i =& n^{-1}\mathbf{PwM}_{\psi^{-1}}(\mathbf{NTT}_{\omega^{-1}}(a))_i \\ =& n^{-1}\psi^{-i} \sum_{j=0}^{n-1} a_j \omega^{-ij} \bmod q \end{aligned} \tag{3}$$

The negNTT algorithm executes polynomial-by-polynomial and performs in-place operation based on butterfly operation. The two widely-used butterflies are Cooley-Tukey (CT) butterfly and GS butterfly as shown in Fig. 1. They share the same time-efficiency and consist of the same elements. CT butterfly performs multiplication before addition and subtraction, while GS butterfly uses a multiplication result only in one path.

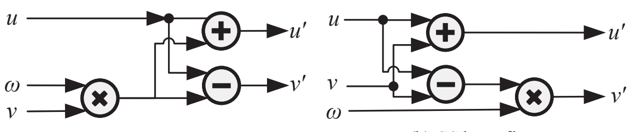

Fig. 1. Comparison between the butterfly structures

III. Optimization of the operations

A. Optimizing Arithmetic Operations over Polyvec

In Kyber algorithm, operations over $\mathrm{polyvec}$ include forward and inverse negNTT , point-wise multiplication, and $\mathrm{polyvec}$ addition, etc. These operations are the most expensive parts in terms of area and time. A representative formula is $\mathbf{As}+\mathbf{e}$, which consists of all the typical $\mathrm{polyvec}$ operations. Since the elements in polynomial matrix $\mathbf{A}$ are sampled in NTT domain, all we need to do in this formula are the following 10 steps [9].

1) Initialize coefficients into a RAM through a write port. This costs $n \times k$ cycles.
2) Calculate $\mathbf{PwM}_{\psi^i}(\mathbf{s})$, which costs $n \times k$ cycles.
3) Transform $\mathrm{polyvec}$ $\mathbf{s}$ to NTT domain by $\mathbf{NTT}_\omega(\mathbf{PwM}_\psi(\mathbf{s}))$. This costs $n \times k \times \log(n)$ cycles.
4) Rearrange the bit-reversed order to normal order which costs about $n \times k$ cycles.
5) Point-wise-multiply $\mathbf{A} \circ \mathbf{negNTT}_\omega(\mathbf{s})$. This costs $n \times k^2$ cycles for multiplication and $n \times k \times (k-1)$ cycles for additions.
6) Perform $\mathbf{NTT}_{\omega^{-1}}$ transformation on the point-wise-multiply result which costs $n \times k \times \log(n)$ cycles.
7) Rearrange the bit-reversed order to normal order again. This costs about $n \times k$ cycles.
8) Calculate $\mathbf{PwM}_{\psi^{-i}}$, which costs $n \times k$ cycles.
9) Calculate $\mathbf{PwM}_{7651}$ to obtain $\mathbf{As}$. This costs $n \times k$ cycles.
10) Add the error $\mathrm{polyvec}$ $\mathbf{e}$ to $\mathbf{As}$, which costs $n \times k$ cycles.

Many hardware implementations utilized CT butterfly structure such as [10], [12], [13]. In a software implementation, Pöppelmann [20] used CT butterfly in forward NTT computing and GS butterfly in inverse NTT computing to avoid pre- and post-operations. However, the latter strategy has side effects in hardware on account of additional area consumption. In this hardware implementation, we only instantiate a GS butterfly structure to avoid step 8 after the inverse NTT transformation. The utilization of GS butterfly enables the following three optimization techniques. (1) Since only one path relates to the multiplication result, we can perform subtraction operation with the pre-adder of the DSP slice. This reduces the delay and area of LUT-based combinational circuits. (2) In order to further reduce the unnecessary latency, step 2 can be merged to step 1 by performing multiplication before storing the coefficients. (3) The step 9 can be merged into step 10 by converting addition into MADD. The techniques (2) and (3) balance the computational load on the multiplier and $2 \times n \times k$ clock cycles are saved.

Unlike the point-wise multiplication for polynomials [10], the extra addition leads to more time and area consumption in the $\mathrm{polyvec}$ multiplication. In step 5, $\mathrm{Kyber1024}$ needs to perform $4 \times n$ multiplication operations and $3 \times n$ additions. For $\mathrm{Kyber512}$, $2 \times n$ multiplication operations and n additions are required. Step 5 can be substituted with MACC operation of DSP which directly accumulates the results of each multiplication into a built-in registers (i.e., dsp_p register), thus reducing the redundant latency for addition. In general, $n \times k \times (k-1)$ cycles are saved.

In subsection III-B, we will further show a technique to eliminate step 4 and step 7. Thus the process costs only $(2\log(n)+k+2) \times n \times k$ clock cycles after our optimizations, which save a total of 29.4% cycles for $\mathrm{Kyber512}$ and 33.3% cycles for $\mathrm{Kyber1024}$. We reconstitute the above 10 steps to 5 as shown in Table II.

B. Optimizing Memory Access

In the NTT implementation, in-place operation helps to achieve a compact memory utilization. In this case, the butterfly structure needs to read two coefficients and write the other two regardless of the rotation factor ω_n. For BRAMs on FPGAs, a read or write operation takes at least 1 clock cycle. Thus, to read a value and then write in the same position using one port would always need 2 clock cycles. Even though the two ports of one BRAM can be configured as two read ports, two write ports or one read and one write for different addresses, the coefficients can not be read and written in the same clock cycle. As a result, dataflow will get a "bubble" and DSP slices will be idle. On hardware, memory access becomes the bottleneck of efficient implementation. In [16], the implementation for $\mathrm{NewHope-Simple}$ takes 7 clock cycles to perform a butterfly operation, in which 2 extra cycles are occupied by memory access. In [13] the coefficients are stored in two columns in a swapped order, but this requires redundant clock cycles to rearrange the paired coefficients. According to the structural characteristics of $\mathrm{polyvec}$ in Kyber, we propose a dual-column sequential storage structure. The proposed structure enables a pipeline memory access scheme which reads and writes 2 coefficients in each clock without redundant latency.

Our storage is arranged quite concise as shown in Fig. 2. For $\mathrm{polyvec}$ $\mathbf{s} = \begin{bmatrix} \mathbf{s}(0), \mathbf{s}(1) \end{bmatrix}^T$, the polynomial coefficients in $\mathbf{s}(0)$ and $\mathbf{s}(1)$ are sequentially initialized in pair like ($\mathbf{s}_0(0)$, $\mathbf{s}_0(1)$), ($\mathbf{s}_1(0)$, $\mathbf{s}_1(1)$), $\cdots$, ($\mathbf{s}_{255}(0)$, $\mathbf{s}_{255}(1)$). Polynomial $\mathbf{s}(0)$ is stored in the higher 13 columns and $\mathbf{s}(1)$ is stored in the lower 13 columns. In the memory, coefficients with the same serial number are stored in the same address. High-dimensional $\mathrm{polyvecs}$ in $\mathrm{Kyber1024}$ are also applicable by increasing the memory depth. As shown in the Algorithm 1, by disjoining the read and write ports, we access and store coefficients at the same time. The algorithm enables a pair of coefficients belonging to different polynomials to be accessed in parallel and operated apart. With the pipeline architecture in subsection III-C, extra latency of memory access in [16] is avoided and thus dataflow is free of bubble.

The storage scheme further enables the underlying bit-reversed address accessing technique, which is employed to solve the challenge of bit-reversal in $\mathrm{NTT}_{\omega^{-1}}$ step. An in-place forward or inverse NTT algorithm transforms the coefficients arrangement either from normal order to bit-reversed order, or

TABLE II
RECONSTITUTE $\mathbf{As} + \mathbf{e}$ INTO 5 STEPS

Step	Operation	DSP Logic	Times of Mul. Kyber512/1024
INIT	$init(\mathbf{PwM}_{\psi}(\mathbf{s}))$	$dsp\_a \times dsp\_b$	512/1024
NTT_{ω}	$\mathbf{negNTT}_{\omega}(\mathbf{s}) = \mathbf{NTT}_{\omega}(\mathbf{PwM}_{\psi}(\mathbf{s}))$	$(dsp\_d - dsp\_a) \times dsp\_b$	2048/4096
MACC	$\mathbf{MACC} = \mathbf{A} \circ \mathbf{negNTT}_{\omega}(\mathbf{s})$	$dsp\_a \times dsp\_b + dsp\_p$	1024/2048
$\text{NTT}_{\omega^{-1}}$	$\mathbf{As}' = \mathbf{PwM}_{\psi^{-1}}(\mathbf{NTT}_{\omega^{-1}}(\mathbf{MACC}))$[a]	$(dsp\_d - dsp\_a) \times dsp\_b$	2048/4096
MADD	$\mathbf{As} + \mathbf{e} = n^{-1} \circ \mathbf{As}' + \mathbf{e}$	$dsp\_a \times dsp\_b + dsp\_c$	512/1024

[a] Operations in $\text{negNTT}_{\omega^{-1}}$ step is slightly different with inverse negNTT. We denote $\mathbf{As}'$ as a intermediate value without n^{-1} operation.

from bit-reversed order to normal order. As a basic method, the authors in [10], [12] deployed a rearrangement step after the forward and inverse NTT operations. In [12], about n clock cycles are consumed to complete bit-reversal. In this work, GS butterfly-based forward NTT_{ω} step reverses the normal order to bit-reversed order. In the $\text{NTT}_{\omega^{-1}}$ step, we use the inverted bit connection of address line. This counteracts the bit-reversed order in the memory. After $\text{NTT}_{\omega^{-1}}$ step, polynomial coefficients will be naturally restored in normal order. We will show that bit-reversed address connection is much cheaper than rearrange memory data in subsection III-C. This technique estimates the step 4 and 7, thus $2 \times n \times k$ operations are saved.

C. Processor Structure and Pipeline Technique

The prcessor structure shown in Fig. 2 consists of three subunits, namely memory, arithmetic logic unit (ALU) and a Finite State Machine (FSM)-based control unit.

The arithmetic logic is reused in different steps and each operation is followed by a modular reduction operation. We implement a Barret reduction unit as illustrated in [21], [22]. Since this reduction unit replaces the multiplying by a constant with shifts-and-adds using LUTs, it becomes the critical path in the whole architecture. The problem is solved by pipeline. We tried to add one to three level pipelines. In order to maintain a compact area, we finally add one level pipeline to obtain a reasonable high frequency.

Algorithm 1: Optimized NTT with GS butterfly

```
Input: Polynomial $a \in \mathcal{R}_q$, $\psi^i$ and $\psi^{-i}$ stored in $psi[\,]$ and $invpsi[\,]$,
       respectively.
Output: Polynomial $a \in \mathcal{R}_q = \mathbf{NTT}_{\omega}(a)$ or $\mathbf{NTT}_{\omega^{-1}}(a)$.

logt = 8;                                              /* $m = (1 \ll logm)$ */
for ($logm = 0; m < n; m++$) do
    $logt = logt - 1$                                  /* $t = (1 \ll logt)$ */
    for ($i = 0; i < m; i++$) do
        $jFirst = i \ll (logt + 1)$
        $jLast = i \ll (logt + 1) + t$
        for ($j = jFirst; j < jLast; j++$) do
            if $NTT_{\omega}$ then
                $(u1, u2) = a[j]$                      /* $\text{NTT}_{\omega}$ */
                $(v1, v2) = a[j + t]$
                $a[j] = (u1 + v1, u2 + v2) \mod q$
                $\omega = psi[(j - jFirst) \ll (logm + 1)]$
                $a[j + t] = ((u1 - v2)\omega, (u2 - v2)\omega) \mod q$
            else
                $(u1, u2) = a[BitReverse(j)]$          /* $\text{NTT}_{\omega^{-1}}$ */
                $(v1, v2) = a[BitReverse(j + t)]$
                $a[BitReverse(j)] = (u1 + v1, u2 + v2) \mod q$
                $\omega^{-1} = invpsi[(j - jFirst) \ll$
                  $(logm + 1) + 1 \ll (7 - logt)]$
                $a[BitReverse(j + t)] =$
                  $((u1 - v2)\omega^{-1}, (u2 - v2)\omega^{-1}) \mod q$
            end
        end
    end
end
```

An external selection signal controls the working stage of the processor. The FSM controls the circuit through the address registers and the multiplexer in ALU. We integrate the mentioned functions into one processor. For NTT computation, our pipeline implementation completes one butterfly in 7 stages (stage 0 to 6) so that the total latency is 2055 clock cycles. Operations in each stage are shown as follows. Note that register $read\_addr\_s$ is initialized to 0.

- **Stage 0 to 1.** Since we enable the read pipeline of RAM_s, latency of read operation is 2 cycles and the first output coefficient pair is $(\mathrm{s}_0(0), \mathrm{s}_0(1))$. The $read\_addr\_s$ updates every clock cycle and the second address would be 128.
- **Stage 2.** Coefficients ($\mathrm{s}_0(0)$ and $\mathrm{s}_0(1)$) are stored in the H_i and L_i registers respectively.
- **Stage 3 to 4.** The coefficients $\mathrm{s}_{128}(0)$, $\mathrm{s}_0(0)$ in H_i and corresponding ω perform a butterfly-and-mod q operation. Coefficient $\mathrm{s}_0(1)$ shifts from L_i to H_i, and $\mathrm{s}_{128}(1)$ is stored in L_i.
- **Stage 5.** Results $adder\_o$ and $barret\_o$ are stored in H_o and L_o, respectively. Note that $adder\_o$ is to be writen into $\mathrm{s}_0(0)$ and $barret\_o$ is to be writen into $\mathrm{s}_{128}(0)$.
- **Stage 6.** Data $\mathrm{s}_0(0)$ stored in H_o and the $adder\_o$ for $\mathrm{s}_0(1)$ are writen into the address 0 in pair. $\mathrm{s}_{128}(0)$ in L_o is shifted to H_o and $barret\_o$ is stored in L_o. They would be writen into address 128 in the next clock cycle.

The processor can also execute other steps in $\mathbf{As} + \mathbf{e}$ by reusing the arithmetic and logic units.

IV. IMPLEMENTATIONS AND COMPARISON

A. Implementations on FPGAs

We have implemented our processor on Xilinx XC7A200T and XC6SLX45T FPGAs with parameter sets (n, k, q) of Kyber512 (256, 2, 7681) and Kyber1024 (256, 4, 7681), respectively. Since the polyvec coefficients need a mass of storage, it is advisable to store data in BRAMs. Each 36Kb BRAM slice in Xilinx 7 series FPGAs can be divided into two 18Kb slices. The 18Kb storage can be configured as 512×36 or 1024×18. Correspondingly, 1 BRAM slice in Spartan6 has 18Kb space. In Kyber512, since the ceiling of each coefficient is $\lceil \log_2 7681 \rceil = 13$ bits width, each polyvec can be storaged in a block of 256×26 memory exactly as simple dual-port mode.

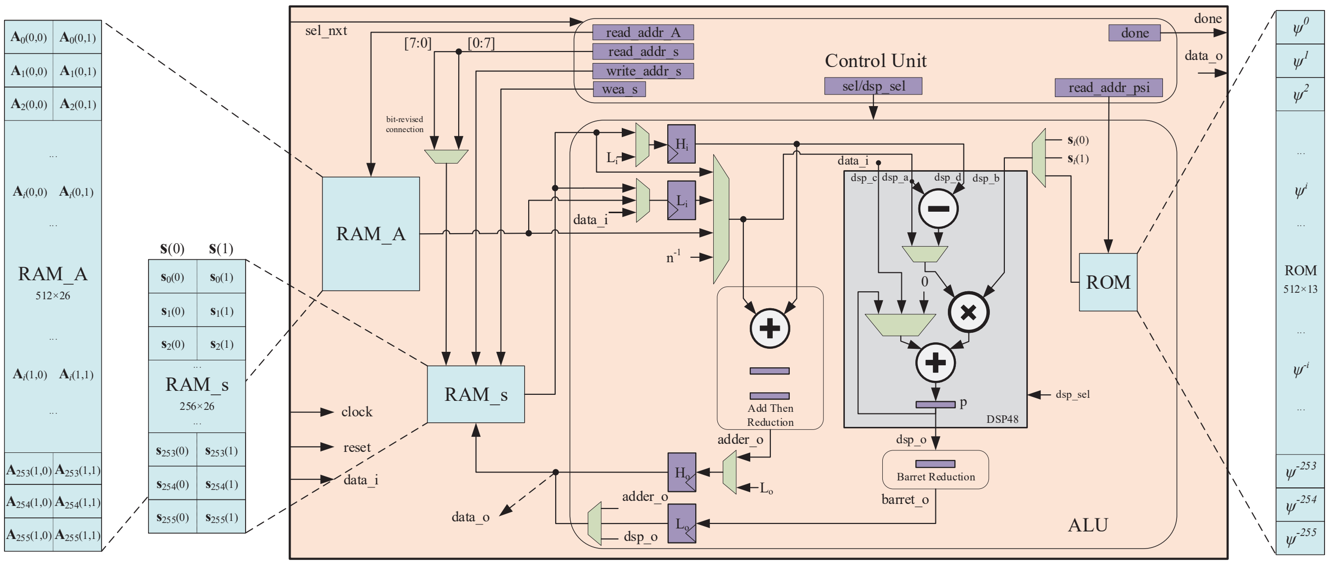

Fig. 2. Processor architecture for vector of polynomials

B. Experimental Results and Comparisons with the Counterparts

The test results obtained after post-place and route (Post-PAR) are listed in Table III. On Artix7, the pipeline Barret reduction module occupies 58 LUTs and 14 FFs and the complex control unit costs 251 LUTs and 118 FFs, in other words, about half the area. The maximum frequency of our processor is 136MHz for $\mathrm{Kyber512}$ and 130MHz for $\mathrm{Kyber1024}$.

Our $\mathrm{polyvec}$ processor is comparable with the polynomial oriented modules for Ring-LWE-based algorithms such as $\mathrm{NewHope}$. For a fair comparison, $\mathrm{Kyber512}$ is to be compared with $\mathrm{NewHope512}$ $(512, 1, 12289)$ and $\mathrm{Kyber1024}$ is to be compared with $\mathrm{NewHope1024}$ $(512, 1, 12289)$ with the similar security level. Some antecedent implementations for polynomial multiplication are also compared because of their similar computational costs. It consumes $\frac{n \times k}{2} \log n = 2304$ numeral multiplication operations for parameter sets $(n, k) = (512, 1)$ and 2048 for $(256, 2)$ in NTT transformation. Most polynomial multiplication (abbreviated as **Mul**) implementations consider the total latency as the sum 2 forward NTT, 1 point-wise multiply and 1 inverse NTT, which costs 7424 multiplication operations in total. Correspondingly, we define **Mul′** for Kyber as the complexity of cascading $2\times$ NTT_ω step, $1\times$ MACC step and $1\times$ $\mathrm{NTT}_{\omega^{-1}}$ step, which costs 7168 multiplication operations after optimizations. In order to compare the performance more intuitively, we calculate the number of NTT and **Mul′**/**Mul** operations per second.

The implementation of $\mathrm{NewHope-Simple}$ in [16] uses one DSP for NTT and another DSP for multipurpose. Although they use CT butterfly for forward NTT and GS butterfly for inverse NTT to avoid $\mathcal{O}(n)$ cycles for rearranging, the time-consuming reduction module and extra memory access cycles are massive burden. Their butterfly operation costs 1 clock cycle on multiplication and leaves 4 for modular reduction and 2 for memory access so that total of $7\times\frac{n}{2}\log(n)$ are used. Their modular reduction module is not further optimized and brings high latency. Although the modular reduction module can be improved, our processor still has a 3x advantages in speed. The LUTs and FFs results for [16] is obtained by re-synthesizing their open source code in Vivado 2018.2. Since the area of different modules can not be simply added up, we only count the independent NTT module. Their total area would be larger than our implementation.

Although we use only one DSP to make the circuit compact, the performance is comparable to those of high performance implements. The implementation of $\mathrm{NewHope}$ in [17] tries to improve performance by adding up to 4 parallel paths. As the cost of high-speed, 8 DSP slices is occupied in NTT and reduction module. The utilization of DSPs is obviously inferior to our implementation. Actually, as a better trade-off, our processor obtain over 55.2% performance with less than their 17.2% logical units.

Compared with earlier Ring-LWE implementations [10], [15], we also have at least 1.89x advantages in terms of efficiency. The advantage is relatively small because the modules for Ring-LWE do not implement the functions like MACC, MADD, etc. The result of [13] is not listed in the table because the area of NTT module is not available.

V. Conclusion and Future Work

In this work, we propose an optimized hardware implementation for Kyber. A processor is realized on FPGAs with a good trade-off between area and performance. The optimized NTT algorithm with GS butterfly cuts off the clock cycles by about one third compared with the textbook implementations. The dual-column sequential storage scheme keep the datapath free of bubble. Besides, the pipeline implementation reuses most logic units so that the structure is kept compact. Taking advantages of the optimizations above, our processor is at least 3x more efficient than the $\mathrm{NewHope}$ implementations. In the future, we will focus on implementing entire Kyber cryptosystem against side channel attack and comparing it with other selected algorithms.

TABLE III
COMPARISON BETWEEN SIMILAR HARDWARE IMPLEMENTATIONS

Processor	**Parameters**($n/q/k$)	**Device**	**Area**		**Cycles**	**Operations/s**	**Efficiency**[a]
Kyber512	256/7681/2	Artix7	**LUTs:** 442	**DSP:** 1	**NTT:** 2055	**NTT:** 66180	**NTT:** 149.7
this work			**FFs:** 237	**BRAM36:** 1.5	**Mul′:** 7197	**Mul′:** 18897	**Mul′:** 42.8
Kyber512	256/7681/2	Spartan6	**LUTs:** 466	**DSP:** 1	**NTT:** 2055	**NTT:** 41443	**NTT:** 88.9
this work			**FFs:** 237	**BRAM18:** 2	**Mul′:** 7197	**Mul′:** 11833	**Mul′:** 25.4
Ring-LWE	512/65537/1	Spartan6	**LUTs:** 1585	**DSP:** 1	**NTT:** NA	**NTT:** NA	**NTT:** NA
[10]			**FFs:** 1205	**BRAM18:** 4	**Mul:** 10014	**Mul:** 19572	**Mul:** 12.3
Ring-LWE	512/65537/1	Spartan6	**LUTs:** 3259	**DSP:** 1	**NTT:** NA	**NTT:** NA	**NTT:** NA
[15]			**FFs:** 3242	**BRAM18:** 6	**Mul:** 9429	**Mul:** 13787	**Mul:** 4.2
Kyber1024	256/7681/4	Artix7	**LUTs:** 477	**DSP:** 1	**NTT:** 4103	**NTT:** 31684	**NTT:** 66.4
this work			**FFs:** 237	**BRAM36:** 2	**Mul′:** 14365	**Mul′:** 9050	**Mul′:** 19.0
NewHope − Simple	1024/12289/1	Artix7	**LUTs:** 415	**DSP:** 2	**NTT:** 35845	**NTT:** 3487	**NTT:** 9.3
[16]			**FFs:** 251	**BRAM36:** 4	**Mul:** 80909	**Mul:** 1545	**Mul:** 3.7
NewHope	1024/12289/1	Artix7	**LUTs:** 2832	**DSP:** 8	**NTT:** 2616	**NTT:** 57339	**NTT:** 21.9
[17]			**FFs:** 1381	**BRAM36:** 10	**Mul:** NA	**Mul:** NA	**Mul:** NA
Kyber1024	256/7681/4	Spartan6	**LUTs:** 506	**DSP:** 1	**NTT:** 4103	**NTT:** 19109	**NTT:** 37.8
this work			**FFs:** 237	**BRAM18:** 3.5	**Mul′:** 14365	**Mul′:** 5458	**Mul′:** 10.8
Ring-LWE	1024/65537/1	Spartan6	**LUTs:** 1644	**DSP:** 1	**NTT:** NA	**NTT:** NA	**NTT:** NA
[10]			**FFs:** 1241	**BRAM18:** 6.5	**Mul:** 21278	**Mul:** 9399	**Mul:** 5.7

[a] Efficiency is defined as number of operations per second per LUT.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (No. 61602476, No. 61772518, No. 61872357 and No. 61802396), Cryptography Development Foundation of China (No. MMJJ20170205, MMJJ20180113), and Special Foundation for Industrial Transformation and Upgrading (Departmental Budget, No. 0714-EMTC-02-00577).

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] K. Basu, D. Soni, M. Nabeel, and R. Karri, "NIST post-quantum cryptography- a hardware evaluation study," *IACR Cryptology ePrint Archive*, p. 47, 2019.

[3] H. Nejatollahi, N. D. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 129:1–129:41, 2019.

[4] O. Regev, "New lattice based cryptographic constructions," in *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pp. 407–416, 2003.

[5] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pp. 84–93, 2005.

[6] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *EUROCRYPT 2010*, pp. 1–23, 2010.

[7] C. Peikert, "How (not) to instantiate Ring-LWE," in *Security and Cryptography for Networks, SCN*, pp, 2016. 411–430.

[8] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Des. Codes Cryptography*, vol. 75, no. 3, pp. 565–599, 2015.

[13] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact Ring-LWE cryptoprocessor," in *Cryptographic Hardware and Embedded Systems, CHES*, pp. 371–391, 2014.

[9] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM," in *IEEE European Symposium on Security and Privacy, EuroS&P*, pp. 353–367, 2018.

[10] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *LATINCRYPT 2012*, pp. 139–158, 2012.

[11] F. Valencia, A. Khalid, E. O'Sullivan, and F. Regazzoni, "The design space of the number theoretic transform: A survey," in *SAMOS*, pp. 273–277, 2017.

[12] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Selected Areas in Cryptography, SAC*, pp. 68–85, 2013.

[14] C. Du, G. Bai, and X. Wu, "High-speed polynomial multiplier architecture for ring-lwe based public key cryptosystems," in *Proceedings of the 26th edition on Great Lakes Symposium on VLSI, GLVLSI*, pp. 9–14, 2016.

[15] T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced lattice-based signatures on reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems, CHES*, pp. 353–370, 2014.

[16] T. Oder and T. Güneysu, "Implementing the NewHope-Simple key exchange on low-cost FPGAs," in *LATINCRYPT*, 2017.

[17] P.-C. Kuo, W.-D. Li, Y.-W. Chen, Y.-C. Hsu, B.-Y. Peng, C.-M. Cheng, and B.-Y. Yang, "High performance post-quantum key exchange on FPGAs," *IACR Cryptology ePrint Archive*, p. 690, 2017.

[18] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A new hope," in *USENIX Security*, pp. 327–343, 2016.

[19] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "NewHope without reconciliation," *IACR Cryptology ePrint Archive*, p. 1157, 2016.

[20] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers," in *LATINCRYPT* pp. 346–365, 2015.

[21] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *CRYPTO*, pp. 175–186, 1993.

[22] T. Pöppelmann, "Efficient implementation of ideal lattice-based cryptography," Ph.D. dissertation, Ruhr University Bochum, Germany, 2016.