A Family of Scalable Polynomial Multiplier Architectures for Ring-LWE Based Cryptosystems

Chaohui Du and Guoqiang Bai

***Abstract*—Many lattice based cryptosystems are based on the Ring learning with errors (Ring-LWE) problem. The most critical and computationally intensive operation of these Ring-LWE based cryptosystems is polynomial multiplication over rings. In this paper, we exploit the number theoretic transform (NTT) to build a family of scalable polynomial multiplier architectures, which provide designers with a trade-off choice of speed vs. area. Our polynomial multipliers are capable to calculate the product of two n-degree polynomials in about $(1.5n \lg n + 1.5n)/b$ clock cycles, where b is the number of the butterfly operators. In addition, we exploit the cancellation lemma to reduce the required ROM storage. The experimental results on a Spartan-6 FPGA show that the proposed polynomial multiplier architectures achieve a speedup of 3 times on average and consume less Block RAMs and slices when compared with the compact design. Compared with the state of the art of high-speed design, the proposed hardware architectures save up to 46.64% clock cycles and improve the utilization rate of the main data processing units by 42.27%. Meanwhile, our designs can save up to 29.41% block RAMs.**

***Index Terms*—post-quantum cryptography, lattice-based cryptography, Ring-LWE, polynomial multiplication, number theoretic transform, FPGA.**

I. Introduction

THE cryptosystems widely used today like RSA [1] and elliptic curve cryptography (ECC) [2], [3] are based on the hardness of number theoretical problems, such as integer factoring and the elliptic curve discrete logarithm problem. Since there are not any known algorithms that can solve these problems efficiently on classical computers, these cryptosystems with large security parameters are believed to be secure against classical computers. However, Shor's algorithm [4] can solve these problems in polynomial time on a powerful quantum computer. Although there is no powerful quantum computer available today, researchers are dedicated to build one and they have predicted that it might be available in about 15 years [5], [6]. Therefore, it is necessary to develop new cryptosystems that can resist both classical computers and quantum computers, which are known as post-quantum cryptography [7].

Since the seminal work of Ajtai [8], which demonstrates connections between the average-case lattice problems and the worst-case lattice problems, Lattice-based cryptography has emerged as one of the most important candidates for post-quantum cryptography. Its security relies on worst-case computational assumptions in lattices that remain hard for both classical computers and quantum computers. The introduction of *learning with errors* (LWE) [9], [10] provides a basic lattice problem with strong security proofs to build many lattice based cryptosystems. However, the LWE problem tends to be not efficient enough for practical applications. To further improve the efficiency of the LWE problem, Lyubashevsky et al. have introduced an algebraic variant of the LWE problem called *Ring-LWE*, and they have proved that it enjoys very strong hardness guarantees [11]. Ring-LWE has served as the basis of various lattice based cryptosystems, such as public key encryption [12], [13], digital signature [14], fully homomorphic encryption [15] and much more.

The arithmetic operations of these Ring-LWE based cryptosystems are carried out over a polynomial ring. The most critical and computationally intensive operation is polynomial multiplication. In recent years, researchers have exploited the *number theoretic transform* (NTT) [16] to speed up polynomial multiplication, such as [17]–[21]. A fully parallel NTT based polynomial multiplier was used to speed up the Ring-LWE based public key cryptosystems [12] in [17], which improved the throughput by a factor of 316 than the software implementation. The implementation is fast but it consumes a huge number of hardware resources, which results in the cryptosystem with medium security cannot even fit on the largest FPGA of the Virtex 6 family. In [18], Pöppelmann et al. proposed a compact design. One butterfly operator is exploited to compute the NTT and inverse-NTT, which reduces the hardware usage. However, the parallel property of the NTT cannot be exploited. Other compact designs like [19], [20] proposed to generate the constant factors on-the-fly instead of storing them in a ROM, which reduced the ROM overhead of [18]. However, these implementations demand either extra modulo p multipliers or extra clock cycles to generate the constant factors, and the parallel property of the NTT cannot be exploited. In order to take advantage of the parallel property of the NTT, Chen et al. proposed an architecture consisted of two butterflies together with two modulo p multipliers in [21]. During the calculation of polynomial multiplication, these processing units compute the two NTT computations concurrently and they are reused to calculate the one inverse-NTT computation. The design achieves a speedup of approximately 3.5 times than the design in [18]. However, the utilization rate of the main data processing units (four integer multipliers and four modulo p modules) is rather low, which is around 52.92%. Therefore, a further speedup of polynomial multiplication is

C. Du is with the Department of Computer Science and Technology, Tsinghua University, Beijing, China, email: dch11@mails.tsinghua.edu.cn.

G. Bai is with the Institute of Microelectronics, Tsinghua University, Beijing, China, email: baigq@tsinghua.edu.cn.

This work was supported by the National Natural Science Foundation of China (Grants 61472208 and U1135004), and by the National Key Basic Research Program of China (Grant 2013CB338004).

possible. Besides, the parallel property of NTT can be further exploited.

In order to facilitate hardware designs of the Ring-LWE based cryptosystems, a family of scalable and efficient polynomial multiplier architectures would be of great help, which provide designers with a trade-off choice of speed vs. area. In this paper, we exploit the *number theoretic transform* [16] and the *negative wrapped convolution theorem* [22] to build a family of scalable and efficient polynomial multiplier architectures. Our proposed polynomial multiplier architectures use b butterfly operators to exploit the parallel property of NTT, where b is a power of 2. To calculate the product of two n-degree polynomials, the proposed architectures take around $(1.5n \lg n + 1.5n)/b$ clock cycles. In addition, we exploit the cancellation lemma [23] to reduce the number of constant factors that are required to store in the ROM. In order to compare with the compact design in [18] and the high-speed design in [21], we have implemented the proposed architectures on a Spartan-6 FPGA. The experimental results show that our polynomial multiplier ($b = 2$) results in a speedup of 3 times on average when compared with the compact design in [18]. Meanwhile, it saves 24.23% up to 36.63% slices and 20.00% up to 30.77% block RAMs. When compared with the state of art of high-speed design [21], our polynomial multiplier ($b = 4$) can save 39.51% up to 46.64% clock cycles and improve the utilization rate of the main data processing units by about 42.27%. Meanwhile, it is able to reduce 17.65% up to 29.41% block RAMs.

The work in this paper is based on our previous work [24] which presents a preliminary version of the proposed polynomial multiplier architectures. The contribution of this work extends the previous work by providing more details and further optimizations. In this paper, we further optimize the polynomial multiplication scheme in algorithm level. Then we introduce several optimizations to reduce the hardware resources. Based on these techniques, we propose faster and tighter hardware architectures. The designs in this paper outperform our previous work by a factor of 1.21 on average. Meanwhile, the designs consume less hardware resources. We also provide implementation results using the parameter sets proposed in [21]. They show that the proposed hardware architectures outperform the state of the art of high-speed design in the literature.

The rest of the paper is organized as follows. In Section II we provide a brief mathematical background. Our proposed scalable and efficient polynomial multiplier architectures are described in Section III. The experimental results are illustrated in Section IV. Finally, we conclude this paper in Section V.

II. Mathematical Preliminaries

In this section, we briefly revisit the notations used through this paper, the Ring-LWE problem, and polynomial multiplication with NTT and the negative wrapped convolution theorem.

A. Notations

In this paper, the base-2 logarithm is denoted lg. We denote by n the degree of the lattice, where n is a power of 2. The

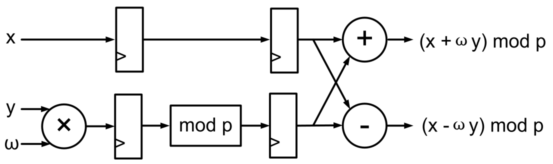


Fig. 1. A butterfly operator. The addition and subtraction are modulo p.

number of butterfly operators is denoted b, where b is a power of 2 and $1 < b \leq n/2$. For any prime p, Z_p represents the ring with the interval $[0, p) \cap Z$. We denote by $Z_p[x]$ the set of polynomials with all coefficients in Z_p. The quotient ring $R_p = Z_p[x]/{<}x^n{+}1{>}$ contains all polynomials of degree less than n in $Z_p[x]$, where p is a prime number that satisfies the condition $p = 1 \bmod 2n$.

B. Ring-LWE

Ring-LWE problem [11] is the building block of various lattice based cryptosystems. It is parameterized by a dimension $n \geq 1$ and a modulus $p \geq 2$, as well as an error distribution, typically a discrete Gaussian distribution D_σ. For a uniformly chosen random ring element $\boldsymbol{s} \in R_p$, the Ring-LWE distribution is sampled by choosing a uniformly random ring element $\boldsymbol{a} \in R_p$ and an error term $\boldsymbol{e}$ from D_σ^n, and outputting the pair $(\boldsymbol{a},\boldsymbol{b}) \in R_p \times R_p$, where $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e} \in R_p$. The goal of the Ring-LWE problem is to distinguish arbitrarily many independent pairs sampled from the Ring-LWE distribution from truly uniform pairs. Lyubashevsky et al. have proved that it is at least as hard as solving certain lattice problems in the worst case [11].

C. Polynomial Multiplication

The arithmetic operations of these Ring-LWE based cryptosystems are carried out over a polynomial ring R_p. Their most critical and computationally intensive operation is polynomial multiplication. Let $\boldsymbol{a} = a_0 + a_1 x + ... + a_{n-1}x^{n-1}$ and $\boldsymbol{s} = s_0 + s_1 x + ... + s_{n-1}x^{n-1}$ be two elements in R_p, the school-book algorithm can be exploited to calculate their product $\boldsymbol{d}$ as follows:

$$\boldsymbol{d} = \boldsymbol{a} \cdot \boldsymbol{s} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i s_j x^{i+j} \ mod \ (x^n + 1). \tag{1}$$

However, the complexity of this scheme is $O(n^2)$.

The *number theoretic transform* (NTT) [16], [22] can speed up polynomial multiplication from $O(n^2)$ to $O(n \lg n)$, which is the *fast Fourier transform* (FFT) [25] defined in a finite field. Algorithm 1 shows the details of iterative NTT algorithm. The primitive n-th root of unity ω_n in Algorithm 1 is defined as the smallest element in Z_p that $\omega_n^n = 1 \bmod p$ and $\omega_n^j \neq 1 \bmod p$ for $0 < j < n$. The core operation of NTT is known as a butterfly operation (line 9 – line 12 of Algorithm 1), which takes two coefficients and a constant factor to calculate their corresponding new values. Figure 1 shows a butterfly operation, which operates on x and y to get the corresponding new coefficients. The bit-reverse operation (line 1 of Algorithm 1) should be performed before calculating

Algorithm 1 Iterative NTT

```
Input: $\boldsymbol{a} \in R_p$, $n$-th primitive root of unity $\omega_n \in Z_p$
Output: $NTT_{\omega_n}(\boldsymbol{a})$
1: $\boldsymbol{A}$ = bit-reverse($\boldsymbol{a}$);
2: $n$ = $\boldsymbol{A}$.length;
3: for ($s = 1$; $s \leq \lg n$; $s = s + 1$) do
4:     $m = 2^s$;
5:     $\omega_m = \omega_n^{n/m}$;
6:     for ($k = 0$; $k < n$; $k = k + m$) do
7:         $\omega = 1$;
8:         for ($j = 0$; $j < m/2$; $j = j + 1$) do
9:             $t = \omega\ \boldsymbol{A}[k + j + m/2] \bmod p$;
10:            $u = \boldsymbol{A}[k + j]$;
11:            $\boldsymbol{A}[k + j] = u + t \bmod p$;
12:            $\boldsymbol{A}[k + j + m/2] = u - t \bmod p$;
13:            $\omega = \omega\ \omega_m \bmod p$;
14:        end for
15:    end for
16: end for
17: return $\boldsymbol{A}$;
```

$\lg n$ stages of butterfly operations. It stores the coefficient at address $addr$ to address bit-reverse($addr$).

Let $\boldsymbol{a'} = (a_0, ..., a_{n-1}, 0, ..., 0)$ and $\boldsymbol{s'} = (s_0, ..., s_{n-1}, 0, ..., 0)$ be vectors of length $2n$, which are the coefficient vectors of $\boldsymbol{a}$ and $\boldsymbol{s}$ padded with 0s respectively. Let ω_{2n} be the primitive $2n$-th root of unity. Then $\boldsymbol{d}$ can be determined as follows:

$$\boldsymbol{d} = NTT^{-1}_{\omega_{2n}}(NTT_{\omega_{2n}}(\boldsymbol{a'}) \odot NTT_{\omega_{2n}}(\boldsymbol{s'}))\ mod\ (x^n + 1), \tag{2}$$

where $\odot$ denotes the component-wise multiplication and the inverse-NTT $NTT^{-1}_{\omega_{2n}}$ is determined by using ω_{2n}^{-1} in Algorithm 1 and multiplying each element of the result by $(2n)^{-1}$ over Z_p [22]. Algorithm 2 shows the details of the iterative inverse-NTT. The scheme based on (2) is much faster than the schoolbook algorithm. However, it requires to double the input of NTT and modulo $(x^n + 1)$.

Algorithm 2 Iterative inverse-NTT

```
Input: $\boldsymbol{a} \in R_p$, $n$-th primitive root of unity $\omega_n \in Z_p$, $\omega_n^{-1}$
Output: $NTT^{-1}_{\omega_n}(\boldsymbol{a})$
1: $\boldsymbol{A}$ = $NTT_{\omega_n^{-1}}(\boldsymbol{a})$;
2: for ($i = 0$; $i < n$; $i = i + 1$) do
3:     $\boldsymbol{A}[i] = \boldsymbol{A}[i]n^{-1} \bmod p$;
4: end for
5: return $\boldsymbol{A}$;
```

We can exploit the *negative wrapped convolution theorem* [22] to further optimize NTT-based polynomial multiplication. Let ω_n be a primitive n-th root of unity in Z_p and $\psi^2 = \omega_n \bmod p$. Let $\hat{\boldsymbol{a}} = (a_0, \psi a_1, ..., \psi^{n-1}a_{n-1})$, $\hat{\boldsymbol{s}} = (s_0, \psi s_1, ..., \psi^{n-1}s_{n-1})$, $\hat{\boldsymbol{d}} = (d_0, \psi d_1, ..., \psi^{n-1}d_{n-1})$ be vectors of length n, which are the coefficient vectors of $\boldsymbol{a}$, $\boldsymbol{s}$, $\boldsymbol{d}$ component-wise multiplied with n-degree vector $(\psi^0, \psi^1, \cdots, \psi^{n-1})$ respectively. The negative wrapped convolution theorem states that

$$\hat{\boldsymbol{d}} = NTT^{-1}_{\omega_n}(NTT_{\omega_n}(\hat{\boldsymbol{a}}) \odot NTT_{\omega_n}(\hat{\boldsymbol{s}})). \tag{3}$$

The i-th coefficient of $\boldsymbol{d}$ can be determined by multiplying the i-th coefficient of $\hat{\boldsymbol{d}}$ with ψ^{-i} over Z_p. Algorithm 3 shows the details of polynomial multiplication using NTT and the negative wrapped convolution theorem. Compared with the scheme based on (2), Algorithm 3 eliminates the modulo $(x^n + 1)$ and reduces the degree of NTT, inverse-NTT, and component-wise multiplication from $2n$ to n.

Algorithm 3 Polynomial multiplication using NTT and the negative wrapped convolution theorem

```
Input: $\boldsymbol{a}, \boldsymbol{s} \in R_p$, $n$-th primitive root of unity $\omega_n \in Z_p$, $\psi$
Output: $\boldsymbol{d} = \boldsymbol{a} \cdot \boldsymbol{s} \in R_p$
1: for ($i = 0$; $i < n$; $i = i + 1$) do
2:     $\hat{\boldsymbol{a}}[i] = \boldsymbol{a}[i]\psi^i \bmod p$
3:     $\hat{\boldsymbol{s}}[i] = \boldsymbol{s}[i]\psi^i \bmod p$
4: end for
5: $\tilde{\boldsymbol{a}}$ = NTT$_{\omega_n}(\hat{\boldsymbol{a}})$
6: $\tilde{\boldsymbol{s}}$ = NTT$_{\omega_n}(\hat{\boldsymbol{s}})$
7: for ($i = 0$; $i < n$; $i = i + 1$) do
8:     $\tilde{\boldsymbol{d}}[i] = \tilde{\boldsymbol{a}}[i]\tilde{\boldsymbol{s}}[i] \bmod p$
9: end for
10: $\hat{\boldsymbol{d}} = NTT^{-1}_{\omega_n}(\tilde{\boldsymbol{d}})$
11: for ($i = 0$; $i < n$; $i = i + 1$) do
12:    $\boldsymbol{d}[i] = \hat{\boldsymbol{d}}[i]\psi^{-i} \bmod p$
13: end for
14: return $\boldsymbol{d}$;
```

III. Scalable and Efficient Polynomial Multiplier Architectures

In this section, we first introduce the techniques to optimize Algorithm 3. Then our scalable and efficient polynomial multiplier architectures are presented, which are based on the optimized scheme. Finally, we illustrate the techniques to optimize the required ROM storage.

A. Algorithm Level Optimization

For simplicity, we assume one butterfly operation or a modulo p multiplication can be calculated in one clock cycle. Since each NTT contains $\lg n$ stages of butterfly operations and a butterfly operator takes 2 out of the total n coefficients to compute their corresponding new values (line 9 – line 12 of Algorithm 1), a NTT computation consumes about $(\lg n \times 0.5n)$ clock cycles. As shown in Algorithm 2, a inverse-NTT computation requires n more clock cycles than a NTT computation to multiply each coefficients with n^{-1}. Carefully examining Algorithm 3, we can find that it requires to calculate component-wise multiplication four times (line 2,3,8,12), NTT twice (line 5,6), and inverse-NTT once (line 10). Therefore, Algorithm 3 demands about $(4 \times n + 2 \times (\lg n \times 0.5n) + (\lg n \times 0.5n + n) = 5n + 1.5n \lg n)$ clock cycles to calculate the product of two n-degree polynomials. However, it can be further optimized to speed up polynomial multiplication.

In [20], the authors proposed to merge the pre-computation (line 2,3 of Algorithm 3) into the NTT computation by changing line 7 of Algorithm 1 from $\omega = 1$ to $\omega = \omega_{2m}$. This technique saves around $(2 \times n)$ clock cycles. For more information, we refer the readers to [20].

Another optimization is based on the fact that: (i) the butterfly operator is usually re-used as a modulo p multiplier to perform component-wise multiplications. For instance, by setting x in Figure 1 to 0, the butterfly operator calculates modulo p multiplication of ω and y, since $(0 + \omega y)$ mod p is equal to ωy mod p. (ii) the constant factor ω is equal to 1 during the first stage of butterfly operations of inverse-NTT. Hence, the new coefficients of x and y can be simplified to calculate $(x + y)$ mod p and $(x - y)$ mod p. Notice that, after performing component-wise multiplication (line 8 of Algorithm 3), the butterfly operator calculates the first stage of butterfly operations of inverse-NTT (line 10 of Algorithm 3). In general, they take around $(n+0.5n)$ clock cycles. However, the tasks can be performed in pipeline by configuring the butterfly operator in a way that the integer multiplier and modulo p module are used to compute component-wise multiplication, and the adder and subtractor are used to calculate the first stage of butterfly operations of inverse-NTT. Therefore, the tasks can be performed in only n clock cycles.

In Algorithm 3, we should multiply each element with n^{-1} at the end of inverse-NTT computation (line 10), and multiply the i-th element of the result with constant factor ψ^{-i} (line 12). In general, two component-wise multiplications are required. However, we can merge the post-computation in Algorithm 3 (line 12) with the component-wise multiplication in Algorithm 2 (line 3) by simply multiplying the i-th element with constant factor $\psi^{-i}n^{-1}$ at the end of inverse-NTT computation. Only one component-wise multiplication is needed, and about n clock cycles can be saved.

To sum up, the above optimization techniques can save $(2n + 0.5n + n)$ clock cycles. As a result, the required clock cycles of a sequential polynomial multiplication is reduced from $(5n + 1.5n \lg n)$ to $(1.5n + 1.5n \lg n)$.

B. Scalable Polynomial Multiplication Architecture

To further speed up polynomial multiplication, we can exploit the parallel property of NTT. During s-th stage of butterfly operations of NTT or inverse-NTT computations, the n coefficients are divided into $n/2$ disjoint subsets, where the i-th coefficient and the $(i$+$2^{s-1})$-th coefficient are in the same subset. Each butterfly operation operates on the two coefficients from a subset to get their corresponding new values. Therefore, there is no data dependence during each stage of butterfly operations, and we can exploit b butterfly operators to operate on b subsets in parallel. For simplicity, we suppose a butterfly operator is able to calculate the new values of a subset in one clock cycle. Then, b butterfly operators can reduce the required clock cycles of a whole stage of butterfly operations from $n/2$ to $\lceil \frac{n/2}{b} \rceil$, where $b \le n/2$. Since n is a power of 2, in order to improve the utilization rate of these butterfly operators, b should be a power of 2. Thus, each stage of butterfly operations can be calculated in $\frac{n/2}{b}$ clock cycles. Since the butterfly operators can be re-used to compute component-wise multiplications and there is no data dependence during the computation of component-wise multiplications, b butterfly operators can work in parallel to reduce the required clock cycles from n to $\frac{n}{b}$. By examining Algorithm 3, we can find that it consists of NTT computations, inverse-NTT computations and component-wise multiplications. Therefore, b butterfly operators can work in parallel to speed up polynomial multiplication by a factor of about b. Based on these observations, we propose to exploit b butterfly operators to build a family of scalable and efficient polynomial multiplier architectures, where b is a power of 2 and $1 < b \le n/2$.

Our proposed polynomial multiplier architectures consist of two RAM modules, ROM modules, b configurable butterfly operators, a configurable network module (CNM), and a controller. Figure 2 shows the datapath of our polynomial multiplier with four butterfly operators. The two RAM modules are used to store the initial coefficients, the intermediate coefficients and the final results. Each RAM module is composed of $b/2$ banks, and each bank is built with simple dual-port RAMs, which can be read and written concurrently. A bank of each RAM module has n/b cells and each cell can store $2\lceil \lg p \rceil$ bits. The ROM modules are used to store the constant factors. We will show the details of our ROM storage scheme in next section. The b configurable butterfly operators are used to calculate butterfly operations or component-wise multiplications in parallel, which are shown in Figure 3. The CNM is used to buffer and shuffle the intermediate coefficients. Figure 4 shows the CNM used in our polynomial multiplier with four butterfly operators ($b = 4$). The datapath is driven by the controller, which generates the addresses and write signals for the RAM modules, addresses for the ROMs, the load and selection signals for the configurable butterfly operators and the CNM.

Before illustrating how the proposed polynomial multiplier architectures perform polynomial multiplication, we introduce the notations used in this section. Since each RAM module in the proposed architectures is composed of $b/2$ banks, and each bank has n/b cells, the physical RAM address can be represented by $L = \lg(n/b)$ bits and the bank index can be represented by $W = \lg(b/2)$ bits. We denote the physical RAM address as $A_{L-1}A_{L-2} \cdots A_1A_0$, where A_i is either '0' or '1'. The bank index is denoted by $B_{W-1}B_{W-2} \cdots B_1B_0$, where B_i is '0' or '1'. Since each cell can store $2\lceil \lg p \rceil$ bits, the cell can be logically partitioned into two *channels*, where each channel contains $\lceil \lg p \rceil$ bits. We use 1-bit C to indicate which channel the coefficient is stored. Therefore, the location of a coefficient can be represented by $A_{L-1}A_{L-2} \cdots A_1A_0B_{W-1}B_{W-2} \cdots B_1B_0C$, which is called the *storage address*. We denote by $a_{i,j}$ the j-th coefficient before performing i-th stage of butterfly operations, where j is called the *virtual address*. And we use a *mapping pattern* to map the virtual address to the storage address. Figure 5 shows an example about how to map the virtual address of $a_{2,j}$ to the storage address $A_2A_1A_0B_0C$. For instance, to locate $a_{2,20}$, whose virtual address is $j = 20$ in decimal and $j = 10100$ in binary, we use the mapping pattern $A_2A_1B_0CA_0$ to figure

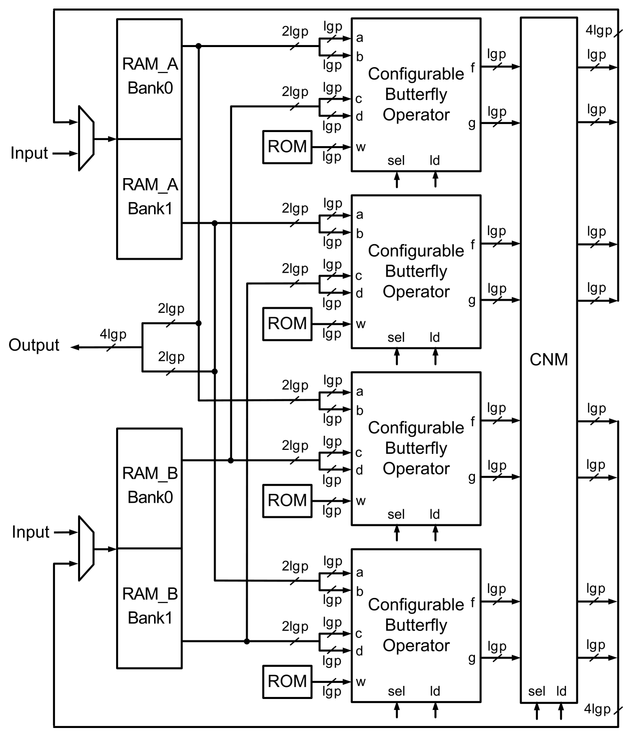
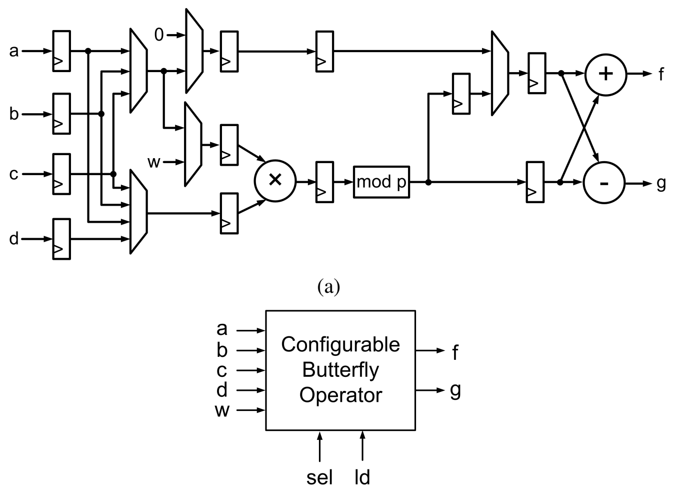


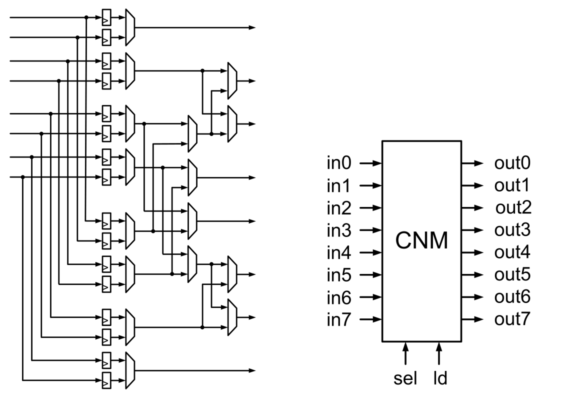
Fig. 2. The datapath of the proposed polynomial multiplier architecture (b=4). CNM represents configurable network module.



(a)

(b)

Fig. 3. The configurable butterfly operator. The addition and subtraction are modulo p. (a) Schematic. (b) Block diagram.



(a) (b)

Fig. 4. The configurable network module used in our design (b=4). (a) Schematic. (b) Block diagram.

	Physical RAM Address	0	1	2	3	4	5	6	7
Bank0	C=0	$a_{2,0}$	$a_{2,1}$	$a_{2,8}$	$a_{2,9}$	$a_{2,16}$	$a_{2,17}$	$a_{2,24}$	$a_{2,25}$
	C=1	$a_{2,2}$	$a_{2,3}$	$a_{2,10}$	$a_{2,11}$	$a_{2,18}$	$a_{2,19}$	$a_{2,26}$	$a_{2,27}$
Bank1	C=0	$a_{2,4}$	$a_{2,5}$	$a_{2,12}$	$a_{2,13}$	$a_{2,20}$	$a_{2,21}$	$a_{2,28}$	$a_{2,29}$
	C=1	$a_{2,6}$	$a_{2,7}$	$a_{2,14}$	$a_{2,15}$	$a_{2,22}$	$a_{2,23}$	$a_{2,30}$	$a_{2,31}$

(a)

Virtual Address	j_4	j_3	j_2	j_1	j_0
Mapping Pattern	A_2	A_1	B_0	C	A_0
Storage Address	j_4	j_3	j_0	j_2	j_1

(b)

Fig. 5. Memory address mapping example (n=32, b=4). (a) The contents of physical memory, where $a_{2,j}$ represents the j-th coefficient before performing second stage of butterfly operations. (b) Mapping pattern $A_2A_1B_0CA_0$ maps virtual address $j_4j_3j_2j_1j_0$ to storage address $A_2A_1A_0B_0C = j_4j_3j_0j_2j_1$. The addresses are represented in the form of binary numbers.

out that $A_2 = 1$, $A_1 = 0$, $B_0 = 1$, $C = 0$, $A_0 = 0$. Thus, the storage address is $A_2A_1A_0B_0C = 10010$ in binary, which means $a_{2,20}$ is stored at channel 0 of the fourth cell of bank 1.

Now we take a closer look at how the proposed polynomial multiplier architectures perform polynomial multiplications. In order to calculate the product of two n-degree polynomials in about $(1.5n + 1.5n \lg n)/b$ clock cycles, our polynomial multiplier architectures work in pipelined fashion and partition the goal into several consecutive tasks:

task0: calculate the first L stages of butterfly operations of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$.

task1: calculate the next W stages of butterfly operations of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$.

task2: calculate the last stage of butterfly operations of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$.

task3: calculate the component-wise multiplications of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$ by the modulo p multipliers of the butterfly operators, and calculate the first stage of butterfly operations of inverse-NTT by the adders and subtractors.

task4: calculate the next $(L-2)$ stages of butterfly operations of inverse-NTT.

task5: calculate the L-th stage of butterfly operations of inverse-NTT.

task6: calculate next W stages of butterfly operations of inverse-NTT.

task7: calculate the last stage of butterfly operations of inverse-NTT.

task8: calculate the component-wise multiplications by configuring the butterfly operators as modulo p multipliers.

Initially, the coefficients of polynomial $\boldsymbol{a}$ and $\boldsymbol{s}$ are stored at RAM_A and RAM_B respectively. The mapping pattern is $CB_{W-1}B_{W-2}\cdots B_1B_0A_{L-1}A_{L-2}\cdots A_1A_0$.

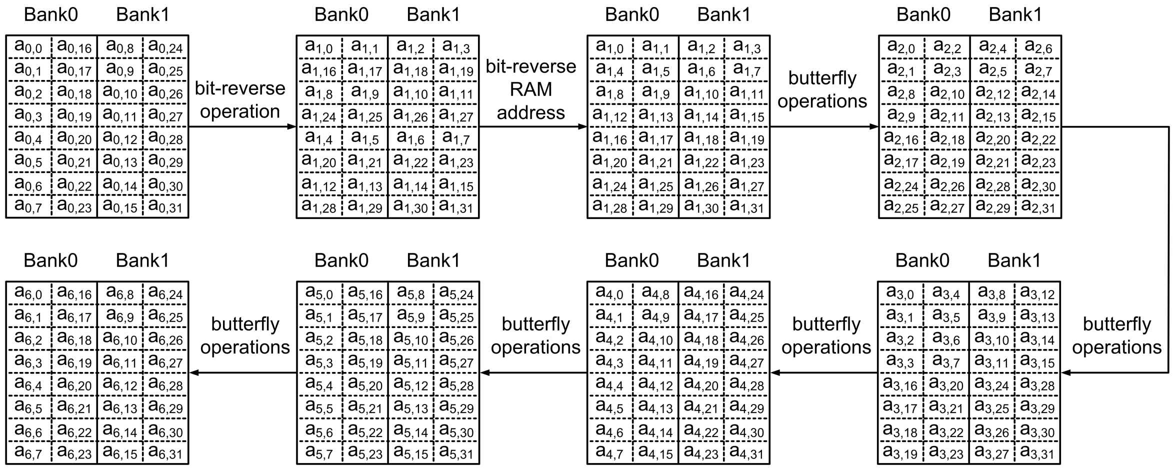


Fig. 6. The contents of RAM_A during the computation of NTT($\hat{\boldsymbol{a}}$) using the proposed polynomial multiplier (n=32, b=4). $a_{0,j}$ represents the initial coefficients of polynomial $\boldsymbol{a}$. $a_{i,j}$ $(0 < i < 6)$ represents the j-th coefficient before performing i-th stage of butterfly operations. $a_{6,j}$ represents j-th coefficient of the final results.

$a_{0,j}$ in Figure 6 shows how the coefficients of polynomial $\boldsymbol{a}$ are initially stored at RAM_A. After bit-reverse operation, the mapping pattern is changed to $A_0A_1\cdots A_{L-2}A_{L-1}B_0B_1\cdots B_{W-2}B_{W-1}C$. Notice that, our polynomial multipliers only change the mapping pattern during bit-reverse operation, they does not need to perform any other operations. Then, our polynomial multipliers bit-reverse the physical memory address, and the mapping pattern is changed to $A_{L-1}A_{L-2}\cdots A_1A_0B_0B_1\cdots B_{W-2}B_{W-1}C$.

During the computation of *task0*, the first L stages of butterfly operations of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$ are calculated in interleaved fashion. Since each RAM module contains $b/2$ banks, and each bank is able to provide 2 coefficients at each clock cycle, b coefficients can be read and/or written at every clock cycle. As $2b$ coefficients from RAM_A or RAM_B are manipulated to perform butterfly operations in interleaved fashion, each RAM module has two clock cycles to load the $2b$ coefficients and store the new $2b$ coefficients back. For instance, our polynomial multipliers load b coefficients from RAM_A at $(2i+j)$-th clock cycle. At $(2i+j+1)$-th clock cycle, b coefficients are read from both RAM_A and RAM_B, and the $2b$ coefficients from RAM_A are operated to perform butterfly operations in parallel. At $(2i+j+2)$-th clock cycle, another b coefficients are loaded from RAM_B, and the $2b$ coefficients from RAM_B are operated by the b butterfly operators. During the computation of s-th $(1 \le s \le L)$ stage of butterfly operations, the i-th, $(i+2^{s-1})$-th, $(i+2^s)$-th, $((i+2^s)+2^{s-1})$-th, $\cdots$, $(i+2^s(b-1))$-th, $((i+2^s(b-1))+2^{s-1})$-th coefficients are loaded from the RAM module in two clock cycles. They are performed the s-th stage of butterfly operations by b butterfly operators in parallel, and the new coefficients are shuffled by the CNM and stored back to the RAM in a way that the i'-th and $(i'+2^s)$-th coefficients are stored at the same cell. After s-th stage of butterfly operations, the mapping pattern is changed to $A_{L-1}\cdots A_{s+1}A_sB_0\cdots B_{W-2}B_{W-1}CA_{s-1}\cdots A_1A_0$. Hence, after performing the first L stages of butterfly operations, the mapping pattern is $B_0B_1\cdots B_{W-2}B_{W-1}CA_{L-1}A_{L-2}\cdots A_1A_0$.

During the calculation of *task1*, the next W stages of butterfly operations of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$ are calculated in parallel. At each clock cycle, b coefficients are provided by RAM_A and operated by $b/2$ out of the total b butterfly operators to perform butterfly operations of $NTT(\hat{\boldsymbol{a}})$, RAM_B also provides b coefficients and the other $b/2$ butterfly operators manipulate these coefficients to perform butterfly operations of $NTT(\hat{\boldsymbol{s}})$. During the computation of s-th $(L < s \le (L+W))$ stage of butterfly operations, at every clock cycle, the i-th, $(i+2^L)$-th, $(i+2\times 2^L)$-th, $(i+3\times 2^L)$-th, $\cdots$, $(i+(b-1)\times 2^L)$-th coefficients are loaded from each RAM module, and they are performed the s-th stage of butterfly operations by $b/2$ butterfly operators in parallel. The new coefficients are shuffled by the CNM and stored back to the RAM in a way that the i'-th and $(i'+2^s)$-th coefficients are stored at the same cell. After s-th stage of butterfly operations, the mapping pattern is changed to $B_0B_1\cdots B_{W-(s-L)-1}CB_{W-(s-L)}\cdots B_{W-1}A_{L-1}\cdots A_1A_0$. So, after performing *task1*, the mapping pattern is $CB_0B_1\cdots B_{W-2}B_{W-1}A_{L-1}A_{L-2}\cdots A_1A_0$.

During *task2*, the last stage of butterfly operations of $NTT(\hat{\boldsymbol{a}})$ and $NTT(\hat{\boldsymbol{s}})$ are calculated in parallel. The butterfly operators work in the similar way with *task1*. However, the new coefficients are shuffled by the CNM and stored back to the RAM in a manner that the i'-th and $(i'+n/2)$-th coefficients are stored at the same cell. The mapping pattern remains $CB_0B_1\cdots B_{W-2}B_{W-1}A_{L-1}A_{L-2}\cdots A_1A_0$.

Before calculating $\lg n$ stages of butterfly operations of inverse-NTT, bit-reverse operation should be performed on all coefficients of $(NTT(\hat{\boldsymbol{a}}) \odot NTT(\hat{\boldsymbol{s}}))$. Based on the fact that $bit-reverse(NTT(\hat{\boldsymbol{a}}) \odot NTT(\hat{\boldsymbol{s}}))$ is equal to $bit-reverse(NTT(\hat{\boldsymbol{a}})) \odot bit-reverse(NTT(\hat{\boldsymbol{s}}))$, we can perform bit-reverse operations before calculating component-wise multiplications. By performing bit-reverse operations, our polynomial multipliers simply change the mapping pattern from $CB_0B_1\cdots B_{W-2}B_{W-1}A_{L-1}A_{L-2}\cdots A_1A_0$ to $A_0A_1\cdots A_{L-2}A_{L-1}B_{W-1}B_{W-2}\cdots B_1B_0C$. Then, our polynomial multipliers bit-reverse the physical RAM address, and the mapping pattern is changed to $A_{L-1}A_{L-2}\cdots A_1A_0B_{W-1}B_{W-2}\cdots B_1B_0C$.

task3 calculates the component-wise multiplication by the modulo p multipliers of the b butterfly operators and calculates the first stage of inverse-NTT by the adders and subtractors. To perform *task3*, we divide the b butterfly operators into two disjoint subsets, each subset contains $b/2$ butterfly operators. For instance, in Figure 2, the upper two butterfly operators are in the same subset, and the bottom two are in another one. At $2i$-th clock cycle, both RAM_A and RAM_B provide b coefficients to the first subset. $2b$ coefficients are provided by RAM_A and RAM_B for the second subset at $(2i+1)$-th clock cycle. Each subset calculates the component-wise multiplications of the $2b$ coefficients in two clock cycles. The products are manipulated by the adders and subtractors to perform the first stage of butterfly operations of inverse-NTT. The new coefficients are shuffled by the CNM and stored back to the RAM in a way that the i'-th and $(i'+2^1)$-th coefficients are stored at the same cell. The mapping pattern is changed to $A_{L-1}A_{L-2}\cdots A_1B_{W-1}B_{W-2}\cdots B_1B_0CA_0$. Notice that, after performing *task3*, our polynomial multipliers store the new coefficients in both RAM_A and RAM_B, and A_{L-1} of the mapping pattern indicates which RAM module the coefficient is stored.

During the computation of *task4*, we treat the coefficients in each RAM module as a $n/2$-degree polynomial. Our polynomial multipliers work in the similar way with *task0* except that the mapping pattern is changed to $A_{L-1}A_{L-2}\cdots A_sB_{W-1}B_{W-2}\cdots B_1B_0CA_{s-1}A_{s-2}\cdots A_0$ after s-th $(1 < s < L)$ stage of butterfly operations. Therefore, after performing *task4*, the mapping pattern is $A_{L-1}B_{W-1}B_{W-2}\cdots B_1B_0CA_{L-2}\cdots A_1A_0$.

During the calculation of *task5*, the b butterfly operators are exploited to calculate the L-th stage of butterfly operations of inverse-NTT. At every clock cycle, the i-th, $(i+2^{L-1})$-th, $(i+2\times 2^{L-1})$-th, $\cdots$, $(i+(b-1)\times 2^{L-1})$-th coefficients are loaded from RAM_A and the $(i+b\times 2^{L-1})$-th, $(i+(b+1)\times 2^{L-1})$-th, $\cdots$, $(i+(2b-1)\times 2^{L-1})$-th coefficients are loaded from RAM_B, the $2b$ coefficients are operated to get their corresponding new values. The new coefficients are shuffled by the CNM, which results in that the i'-th and $(i'+2^L)$-th coefficients are stored at the same cell and the i'-th and $(i'+2^{L-1})$-th coefficients are stored at different RAM modules. Thus, the operands of the rest stages of butterfly operations are from the same RAM module. The mapping pattern is changed to $B_{W-1}B_{W-2}\cdots B_1B_0CA_{L-1}A_{L-2}\cdots A_1A_0$ after the calculation of *task5*.

task6 computes the next W stages of butterfly operations of inverse-NTT. Our polynomial multipliers perform *task6* in the similar way with *task1*. At each clock cycle, $b/2$ butterfly operators calculate butterfly operations on b coefficients from RAM_A, and the other $b/2$ butterfly operators operate on b coefficients from RAM_B. During the computation of s-th $(L < s \leq (L+W))$ stage of butterfly operations, the i-th, $(i+2^L)$-th, $(i+2\times 2^L)$-th, $(i+3\times 2^L)$-th, $\cdots$, $(i+(b-1)\times 2^L)$-th coefficients are manipulated by $b/2$ butterfly operators, and their new coefficients are shuffled by the CNM, which results in the i'-th and $(i'+2^s)$-th coefficients are stored at the same cell. After s-th stage of butterfly operations, the mapping pattern is changed to $B_{W-1}B_{W-2}\cdots B_{s-L}CB_{s-L-1}B_0A_{L-1}A_{L-2}\cdots A_1A_0$. Hence, after performing *task6*, the mapping pattern is $CB_{W-1}B_{W-2}\cdots B_1B_0A_{L-1}A_{L-2}\cdots A_1A_0$.

Our polynomial multipliers calculate *task7* in the similar way with *task2*. At every clock cycle, b coefficients from RAM_A and b coefficients from RAM_B are manipulated to perform the last stage of butterfly operations of inverse-NTT. The new coefficients are shuffled by the CNM and stored back to the RAM in a manner that the i'-th and $(i'+n/2)$-th coefficients are stored at the same cell. The mapping pattern is still $CB_{W-1}B_{W-2}\cdots B_1B_0A_{L-1}A_{L-2}\cdots A_1A_0$, and A_{L-1} indicates which RAM module the coefficient is stored.

During the computation of *task8*, our polynomial multipliers component-wise multiply the i-th coefficient with $\psi^{-i}n^{-1}$ by configuring the b butterfly operators as modulo p multipliers. At each clock cycle, b coefficients are loaded from RAM_A or RAM_B, and they are manipulated by b butterfly operators in parallel. The b new coefficients are shuffled by the CNM and stored back to RAM_A, and the mapping pattern remains $CB_{W-1}B_{W-2}\cdots B_1B_0A_{L-1}A_{L-2}\cdots A_1A_0$.

After these consecutive tasks, our polynomial multipliers finish computing the product of polynomial $\boldsymbol{a}$ and $\boldsymbol{s}$, and the result is stored at RAM_A. Notice that, the final mapping pattern is exactly the same as the initial mapping pattern, which means our polynomial multipliers storing the final coefficients in the same manner as storing the initial coefficients of $\boldsymbol{a}$ and $\boldsymbol{s}$.

C. ROM Storage Scheme

In the proposed polynomial multiplier architectures, we store the constant factors in ROMs instead of generating them on-the-fly. By examining Algorithm 3, we can tell that it requires to store ψ^i $(0 \leq i < n)$ for the pre-computation (line 2,3), $\omega_{2^s}^j$ $(1 \leq s \leq \lg n, 0 \leq j < 2^{s-1})$ for NTT computation (line 5,6), $(\omega_{2^s}^j)^{-1}$ $(1 \leq s \leq \lg n, 0 \leq j < 2^{s-1})$ for inverse-NTT computation (line 10), and ψ^{-i} $(0 \leq i < n)$ for post-computation (line 12). Therefore, around $4n$ constant factors are needed to store in the ROM, where each constant factor takes $\lceil \lg p \rceil$ bits.

However, we can exploit the cancellation lemma [23] to reduce the required ROM storage, which states that $\omega_{dn}^{dk} = \omega_n^k$ for any integers $n \geq 0$, $k \geq 0$, and $d > 0$. Therefore, the following equation can be derived by exploiting the cancellation lemma:

$$\omega_{2^s}^j = \omega_{2^{(\lg n - s)}2^s}^{2^{(\lg n-s)}j} = \omega_n^{(n-2^s)j}, \tag{4}$$

where $1 \leq s \leq \lg n, 0 \leq j < 2^{s-1}$. It can be used to simplify the constant factor storage for NTT computation. Instead of storing $\omega_{2^s}^j$ $(1 \leq s \leq \lg n, 0 \leq j < 2^{s-1})$, we can only store ω_n^j $(0 \leq j < \frac{n}{2})$ and resolve the other ωs from these constant factors by (4). This technique can also be applied to simplify the constant factors for the inverse-NTT computation, which results in that only $(\omega_n^j)^{-1}$ $(0 \leq j < \frac{n}{2})$ are required to store in the ROM.

As ω_n is equal to ψ^2, and ψ^i $(0 \leq i < n)$ are needed to store for the pre-computation, we can further reduce the required ROM storage by resolving ω_n^j $(0 \leq j < \frac{n}{2})$ from ψs instead of storing them in the ROM. Therefore, only ψ^i $(0 \leq i < n)$ are needed to store for the pre-computation and NTT computation.

Since we merge the post-computation in Algorithm 3 with the component-wise multiplication of inverse-NTT computation, we need to store $\psi^{-i}n^{-1}$ $(0 \leq i < n)$. These constant factors are only required during the computation of *task8*, and each factor is operated by one specific butterfly operator. Therefore, we can divide these constant factors into different subsets, and each ROM only requires to store a subset instead of the whole n constant factors.

To sum up, each ROM in the proposed polynomial multiplier architectures needs to store ψ^i $(0 \leq i < n)$, $(\omega_n^j)^{-1}$ $(0 \leq j < \frac{n}{2})$, and a subset of $\psi^{-i}n^{-1}$ $(0 \leq i < n)$. In practical applications, the ROM module can be implemented with dual-port ROMs, where each port is able to perform read operations independently. Therefore, instead of b ROMs, our polynomial multipliers can be implemented with only $b/2$ ROMs.

IV. Experimental Results and Performance

We have implemented the proposed polynomial multiplier architectures with 2/4/8 butterfly operators on the Xilinx Spartan-6 LX100 FPGA. The resource consumptions and the performance results were obtained from the Xilinx ISE 14.7 tool after place and route analysis. Notice that, the proposed architectures support polynomial multiplications for different degree n and moduli p. However, in order to make a fair comparison with the design in [18], we target the same secure parameter sets (n=256/512/1024/2048/4096, p=65537).

Table I shows the resource consumptions and performance of the proposed polynomial multiplier architectures. Recall that our multipliers have two RAM modules, and each module is built with dual-port RAMs. A block RAM on Spartan-6 FPGAs stores up to 18K bits of data, and can be configured as two independent 9Kb RAMs (denoted as 0.5 BRAM) [26]. Therefore, the 9Kb RAMs are configured as simple dual-port RAMs to build the two RAM modules in our polynomial multipliers. They are also configured as dual-port ROMs to build ROM modules in our implementations to store the constant factors. Table I reveals that our polynomial multipliers are capable to perform a n-degree polynomial multiplication in around $(1.5n + 1.5n \lg n)/b$ clock cycles.

The comparison between the compact design in [18] and our polynomial multiplier with two butterfly operators (b=2) is shown in Table II. Our design consumes one more DSP than the design in [18]. However, our implementation saves on average 31.16% slices and 28.85% Block RAMs. On the other hand, our polynomial multiplier reduces around 60.32% clock cycles and achieves a speedup of 3.02 times on average.

Since the proposed polynomial multiplier architectures support polynomial multiplications for different degree n and moduli p, we have also implemented the proposed polynomial multipliers using the secure parameter sets proposed in [21] for different applications. The secure parameter sets are shown in Table III. In order to make a fair comparison with the design in [21], we have implemented the modulo p modules using the same technique presented in [21]. How to choose the secure parameter set and design the modulo p module is out of the scope of this paper, the interested reader might refer to [12], [21] for more information.

TABLE III
SECURE PARAMETER SETS PROPOSED IN [21] FOR DIFFERENT APPLICATIONS.

Application	Degree of polynomial (n)	prime p
LWE [27]	256	1049089
LWE [27]	512	4206593
SHE [28]	1024	536903681
SHE [28]	2048	$2^{57} + 25 \cdot 2^{13} + 1$

Table IV shows the comparison of our polynomial multiplier (b=4) with the high-speed design in [21]. Our implementation takes on average 51.74% more slices than the implementation in [21]. However, our implementation reduces on average 22.12% Block RAMs. Moreover, it saves on average 44.27% clock cycles and speeds up polynomial multiplication by a factor of about 1.76.

Since extra clock cycles are required to fill in the pipeline, and idle clock cycles may be introduced to avoid read/write collision between different stages of butterfly operations, the utilization rate of the integer multipliers of the b butterfly operators cannot achieve 100% in our implementations. The utilization rate of the integer multipliers is defined as follows:

$$utilization\ rate = \frac{RM}{TC\ \times\ NM}\ , \tag{5}$$

where RM is the theoretically total integer multiplications required to perform a polynomial multiplication, TC is the total clock cycles required to perform a polynomial multiplication in the implementation, and NM is the number of integer multipliers in the design. Recall that, Algorithm 3 has been optimized to calculate NTT twice, inverse-NTT once, component-wise multiplication twice, and the first stage of butterfly operations of inverse-NTT are merged into component-wise multiplications. Since NTT demands $(n/2 \times \lg n)$ integer multiplications, inverse-NTT demands $(n/2 \times (\lg n - 1) + n)$ integer multiplications, and component-wise multiplication demands n integer multiplications, $((n/2 \times \lg n) \times 2 + (n/2 \times (\lg n - 1) + n) + n \times 2) = (1.5n \lg n + 1.5n)$ integer multiplications are required to perform a polynomial multiplication. Table V shows the utilization rate of integer multipliers in different implementations. The utilization rate in our design is on average 95.19%, which improves the utilization rate in [21] by about 42.27%. As the modulo p modules are used to reduce the product generated from the integer multipliers to the interval $[0, p)$, its utilization rate is equal to the utilization rate of the integer multipliers. In other words, our design also improves the utilization rate of the modulo p modules by about 42.27%.

V. Conclusion

A family of efficient and scalable NTT based polynomial multiplier architectures are presented for Ring-LWE

TABLE I
IMPLEMENTATION RESULTS OF THE PROPOSED SCALABLE POLYNOMIAL MULTIPLIER ARCHITECTURES ON A SPARTAN-6 LX100 FPGA. THE COLUMN "MUL/S" CONTAINS THE NUMBER OF POLYNOMIAL MULTIPLICATIONS THAT CAN BE CALCULATED PER SECOND.

n	b	Flip-Flop	LUT	Slice	DSP	BRAM	Period (ns)	Cycles	Latency (μs)	Mul/s
256	2	1050	989	394	2	2	4.133	1779	7.353	136006
256	4	1830	1629	695	4	3	4.189	917	3.841	260327
256	8	3420	3073	1183	8	6	4.956	492	2.438	410113
512	2	1076	1042	418	2	2.5	4.136	3895	16.110	62074
512	4	1853	1671	648	4	4	4.292	1977	8.485	117851
512	8	3458	3376	1218	8	8	4.447	1021	4.540	220245
1024	2	1105	1082	445	2	4.5	4.198	8507	35.712	28001
1024	4	1879	1708	717	4	6	4.272	4285	18.306	54628
1024	8	3468	3108	1223	8	12	4.538	2177	9.879	101222
2048	2	1119	1091	448	2	9	4.205	18495	77.771	12858
2048	4	1897	1734	757	4	12	4.211	9281	39.082	25587
2048	8	3503	3276	1317	8	18	4.538	4677	21.224	47115
4096	2	1141	1154	476	2	18	4.133	40003	165.332	6048
4096	4	1920	1793	742	4	24	4.343	20037	87.021	11491
4096	8	3525	3382	1358	8	36	4.408	10057	44.331	22557

TABLE II
COMPARISON OF OUR IMPLEMENTATION (B=2) WITH THE COMPACT DESIGN IN [18] ON A SPARTAN-6 LX100 FPGA.

Design	n	Slice	DSP	BRAM	Period (ns)	Cycles	Cycle Reduction	Latency (μs)	Speedup
Design in [18]	256	520	1	2.5	4.785	4774	–	22.844	–
This work	256	394	2	2	4.133	1779	62.74%	7.353	×3.11
Design in [18]	512	615	1	4	5.102	10014	–	51.091	–
This work	512	418	2	2.5	4.136	3895	61.10%	16.110	×3.17
Design in [18]	1024	659	1	6.5	5.000	21278	–	106.390	–
This work	1024	445	2	4.5	4.198	8507	60.02%	35.712	×2.98
Design in [18]	2048	707	1	12.5	4.902	45326	–	222.188	–
This work	2048	448	2	9	4.205	18495	59.20%	77.771	×2.86
Design in [18]	4096	684	1	25	5.076	96526	–	489.966	–
This work	4096	476	2	18	4.133	40003	58.56%	165.332	×2.96

TABLE IV
COMPARISON OF OUR IMPLEMENTATION (B=4) WITH THE HIGH-SPEED DESIGN IN [21] ON A SPARTAN-6 LX100 FPGA.

Design	n	Slice	DSP	BRAM	Period (ns)	Cycles	Cycle Reduction	Latency (μs)	Speedup
Design in [21]	256	580	16	8.5	3.802	1630	–	6.197	–
This work	256	1075	16	6	3.838	986	39.51%	3.784	×1.64
Design in [21]	512	870	16	8.5	3.946	3630	–	14.323	–
This work	512	1284	16	7	3.900	2001	44.88%	7.804	×1.84
Design in [21]	1024	915	16	14	4.050	7971	–	32.282	–
This work	1024	1412	16	11	4.250	4300	46.05%	18.275	×1.77
Design in [21]	2048	2374	64	50	4.751	17454	–	82.923	–
This work	2048	2842	64	40	4.995	9314	46.64%	46.523	×1.78

TABLE V
UTILIZATION RATE OF THE INTEGER MULTIPLIERS IN OUR IMPLEMENTATION AND THE DESIGN IN [21].

Design	n	Required Multiplications (RM)	Cycles (TC)	Number of Multipliers (NM)	Utilization Rate
Design in [21]	256	3456	1630	4	53.01%
This work	256	3456	986	4	87.63%
Design in [21]	512	7680	3630	4	52.89%
This work	512	7680	2001	4	95.95%
Design in [21]	1024	16896	7971	4	52.99%
This work	1024	16896	4300	4	98.23%
Design in [21]	2048	36864	17454	4	52.80%
This work	2048	36864	9314	4	98.95%

based cryptosystems. The proposed architectures can calculate the product of two n-degree polynomials in about $(1.5n + 1.5n \lg n)/b$ clock cycles, which provide designers with a trade-off choice of speed vs. area. The implementation results of the proposed polynomial multipliers on a Spartan-6 FPGA show that our polynomial multiplier architectures achieve a speedup of 3 times on average while consuming less slices and block RAMs than the compact design. The results also show that our implementation is faster than the state of the art of high-speed design by a factor of about 1.76. Meanwhile, our design reduces on average 22.12% Block RAMs and improves the utilization rate of main data processing units by about 42.27%.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, feb 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[2] V. Miller, "Use of elliptic curves in cryptography," *Advances in Cryptology – CRYPTO '85 Proceedings*, vol. 218, pp. 417–426, 1986. [Online]. Available: http://dx.doi.org/10.1007/3-540-39799-X_31

[3] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comp.*, vol. 48, pp. 203–209, 1987. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-1987-0866109-5

[4] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pp. 124–134, Nov 1994.

[5] K. Chang, "I.B.M. researchers inch toward quantum computer," The New York Times, 2012, http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html?%20r=1&hpw&_r=0.

[6] S. Rich and B. Gellman, "NSA seeks to build quantum computer that could crack most types of encryption," The Washington Post, Jan 2014, http://www.washingtonpost.com/world/national-security/nsa-seeks-to-build-quantum-computer-that-could-crack-most-types-of-encryption/2014/01/02/8fff297e-7195-11e3-8def-a33011492df2_story.html.

[7] D. Bernstein, *Introduction to post-quantum cryptography*, D. Bernstein, J. Buchmann, and E. Dahmen, Eds. Springer Berlin Heidelberg, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88702-7_1

[8] M. Ajtai, "Generating hard instances of lattice problems (extended abstract)," *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, pp. 99–108, 1996. [Online]. Available: http://doi.acm.org/10.1145/237814.237838

[9] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, pp. 84–93, 2005. [Online]. Available: http://doi.acm.org/10.1145/1060590.1060603

[10] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé, "Classical hardness of learning with errors," *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, pp. 575–584, 2013. [Online]. Available: http://doi.acm.org/10.1145/2488608.2488680

[11] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Advances in Cryptology – EUROCRYPT 2010*, vol. 6110, pp. 1–23, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13190-5_1

[12] R. Lindner and C. Peikert, "Better key sizes (and attacks) for lwe-based encryption," *Topics in Cryptology – CT-RSA 2011*, vol. 6558, pp. 319–339, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19074-2_21

[13] R. Lindner and C. Peikert, "Better key sizes (and attacks) for lwe-based encryption," Cryptology ePrint Archive, Report 2010/613, 2010, http://eprint.iacr.org/.

[14] V. Lyubashevsky, "Lattice signatures without trapdoors," *Advances in Cryptology – EUROCRYPT 2012*, vol. 7237, pp. 738–755, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29011-4_43

[15] C. Gentry, "Fully homomorphic encryption using ideal lattices," *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, pp. 169–178, 2009. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[16] J. M. Pollard, "The fast Fourier transform in a finite field," *Math. Comp.*, vol. 25, pp. 365–374, 1971. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-1971-0301966-0

[17] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss, "On the design of hardware building blocks for modern lattice-based encryption schemes," *Cryptographic Hardware and Embedded Systems – CHES 2012*, vol. 7428, pp. 512–529, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_30

[18] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," *Progress in Cryptology – LATINCRYPT 2012*, vol. 7533, pp. 139–158, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33481-8_8

[19] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pp. 81–86, June 2013.

[20] S. Roy, F. Vercauteren, N. Mentens, D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," *Cryptographic Hardware and Embedded Systems – CHES 2014*, vol. 8731, pp. 371–391, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44709-3_21

[21] D. Chen, N. Mentens, F. Vercauteren, S. Roy, R. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 62, no. 1, pp. 157–166, Jan 2015.

[22] F. Winkler, *Polynomial algorithms in computer algebra*, ser. Texts and monographs in symbolic computation. Wien, New York: Springer, 1996. [Online]. Available: http://opac.inria.fr/record=b1091650

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*, 3rd ed. The MIT Press, 7 2009. [Online]. Available: http://amazon.com/o/ASIN/0262033844/

[24] C. Du and G. Bai, "A family of scalable polynomial multiplier architectures for lattice-based cryptography," *Trust, Security and Privacy in Computing and Communications (TrustCom), 2015 IEEE 14th International Conference on*, pp. 392–399, Aug 2015.

[25] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comp.*, vol. 19, pp. 297–301, 1965.

[26] Xilinx, "Spartan-6 fpga block ram resources user guide," 2011, http://www.xilinx.com/support/documentation/user_guides/ug383.pdf.

[27] M. Rckert and M. Schneider, "Estimating the security of lattice-based cryptosystems," Cryptology ePrint Archive, Report 2010/137, 2010, http://eprint.iacr.org/.

[28] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, pp. 113–124, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046682