Compact Hardware Implementation of Ring-LWE Cryptosystems

Sujoy Sinha Roy[1], Frederik Vercauteren[1], Nele Mentens[1], Donald Donglong Chen[2] and Ingrid Verbauwhede[1]

[1]ESAT/SCD-COSIC and iMinds, KU Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
Email: {firstname.lastname}@esat.kuleuven.be

[2]Department of Electronic Engineering,
City University of Hong Kong
Tat Chee Avenue, Kowloon, Hong Kong SAR
Email: donald.chen@my.cityu.edu.hk

Abstract. In this paper we propose an efficient and compact hardware implementation of a polynomial multiplier based on the Fast Fourier Transform (FFT) for use in ring-LWE cryptosystems. We optimize the forward wrapped convolution by merging the pre-processing and the FFT and propose an advanced memory access scheme which reduces the number of memory accesses and the number of RAM slices used in the design. These techniques result in a hardware implementation of a polynomial multiplier for the ring-LWE cryptosystem of dimension 256 that uses only 281 slices and one block RAM on a Virtex V.
Finally, we also propose a modification of a ring-LWE encryption system that reduces the number of FTT operations from five to four resulting in a near 20% speed-up.

Keywords. Lattice-based cryptography, Polynomial multiplication, Hardware implementation, Fast Fourier Transform, Number Theoretic Transform, ring-LWE

1 Introduction

Modern public key cryptography algorithms such as RSA, DSA and ECDSA [8] are based on number theoretic problems which are hard to solve using present-day computers. However in the post-quantum era, these problems are solvable in polynomial time. In order to prevent such potential risks in the future, cryptosystems that are also secure in post-quantum world has gained vast interest from the research world.

Lattice-based cryptography is considered as a potential candidate for quantum-secure public key cryptography due to its wide applicability [15] and its security proofs which are based on worst-case hardness of well known lattice problems. Beside significant progress in theoretical lattice-based cryptography [9, 10, 13],

practical implementations are gaining importance in the research community [1, 4, 5, 11]. The *learning with errors* (LWE) problem [14] has become a foundation of several cryptographic schemes. The ring variant of the LWE problem, known as the ring-LWE [7] is more efficient than LWE and has been used to build several practical public key encryption schemes.

The most computationally intensive operation in the ring-LWE cryptosystem is the multiplication of two polynomials in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$. Hence the implementation of an efficient polynomial multiplier architecture is essential to push forward the use of ring-LWE public key cryptography in practical systems. The Fast Fourier Transform (FFT) is considered as the most efficient algorithm to multiply two polynomials due its quasi-linear complexity $\theta(n \log n)$ [3].

The most important hardware implementations of polynomial multiplication architectures are [1, 5, 11]. In [5], a parallel butterfly structure is used to implement the polynomial multipliers for the ring-LWE cryptosystems of various dimensions. However the polynomial multipliers are very large because of the fully parallel structure. In [11], a sequential polynomial multiplier architecture is designed to use the FPGA resources in an efficient way. The polynomial multiplier uses a dedicated ROM to store all the twiddle factors which are required during the FFT computation. To reduce the area requirement, the polynomial multiplier in [1] does not keep any dedicated ROM and computes the twiddle factors whenever required. The implementation also focuses on increasing the utilization of the computation blocks present in the polynomial multiplier architecture.

Our contributions: In this paper we present a polynomial multiplier architecture based on the Number Theoretic Transform (NTT) algorithm. The polynomial multiplier is designed to have small area and memory requirement, but we also focus on computational optimization to keep the number of cycles small. This paper makes following contributions.

1. During the NTT computation, the coefficients are multiplied by the twiddle factors that are computed using repeated multiplications. In [11] a precomputed table (ROM) is used to avoid this fixed computation cost. A more compact implementation in [1] does not use the ROM and computes the twiddle factors by performing repeated multiplications. In this paper we reduce the number of multiplications by handling the nested loops in a better way.
2. The implementations in [1, 11] uses negative wrapped convolution to reduce the number of evaluations in both forward and backward NTT computations. However, the use of negative wrapped convolution has a pre and post computation overhead. In this paper we propose an efficient pre-computation technique which reduces the cost of the forward NTT.
3. The intermediate coefficients are stored in memory (RAM) during the NTT computation. Access to the RAM is a bottleneck for speeding-up the NTT computation. In the implementations [11, 1], FPGA-RAM slices are placed in parallel to avoid this bottleneck. In this paper we propose an efficient

memory access scheme which reduces the number of RAM accesses, optimizes the number of block RAMs and still achieves maximum utilization of the computational blocks present in the polynomial multiplier.
4. We propose an optimization in the ring-LWE public key cryptosystem. The proposed scheme requires four NTT computations compared to five [12] and thus can achieve nearly 20% reduction in the computation cost.

The remainder of the paper is organized as: In Section 2 we provide a brief mathematical background and our optimization techniques of the NTT are described in Section 3. Section 4 presents our hardware architecture for the polynomial multiplier and Section 5 reports on the experimental results of this implementation. Finally, in Section 6, we propose an optimization of an existing ring-LWE encryption system that reduces the number of NTTS from five to four.

2 Background

In this section we present a brief mathematical background required to understand this paper. Since the contribution of our paper is in optimizing the ring-LWE public key cryptosystem and the polynomial multiplication, we provide a brief background of the ring-LWE cryptosystem and polynomial multiplication using FFT.

2.1 The LWE Problem

The *learning with errors* (LWE) problem is a machine learning problem which is conjectured to be a hard problem. In 2005 Regev [14] proved that solving the LWE problem is equivalent to solving several worst-case lattice problems. Since then, the LWE problem has become a popular basis for developing quantum secure lattice-based cryptosystems.

The LWE problem is parameterized by a dimension $n \geq 1$, an integer modulus $q \geq 2$ and an error distribution (discrete Gaussian distribution) $\mathcal{X}$ over integers. For a uniformly chosen $\mathbf{s} \in \mathbb{Z}_q^n$, the LWE distribution $A_{s,\mathcal{X}}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ consists of the sample points $(\mathbf{a}, t)$ where $\mathbf{a}$ is chosen uniformly from $\mathbb{Z}_q^n$ and $t = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q \in \mathbb{Z}_q$ and e is sampled from the error distribution $\mathcal{X}$. The *search* version of the LWE problem asks to find the $\mathbf{s}$ given a polynomial number of pairs $(\mathbf{a}, t)$ sampled from the LWE distribution $A_{s,\mathcal{X}}$. In the *decision* version of the LWE problem, the solver needs to distinguish with an non-negligible advantage between a polynomial number of samples drawn from $A_{s,\mathcal{X}}$ and the same number of samples drawn from $\mathbb{Z}_q^n \times \mathbb{Z}_q$. For hardness proofs of the search and decision LWE problems, interested readers are referred to [6].

The LWE Public Key Cryptosystem : The initial LWE public key cryptosystem in [14] is based on matrix operations which are quite inefficient and requires a large key size. To achieve computational efficiency and to reduce key size, an algebraic variant of the LWE called *ring*-LWE [7] uses special structured

ideal lattices. Such lattices correspond to ideals in rings $\mathbb{Z}[\mathbf{x}]/\langle f\rangle$, where f is an irreducible polynomial of degree n. For efficiency reason, the ring is taken as $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f\rangle$ and the irreducible polynomial as $f(x) = x^n + 1$, where n is a power of two and the prime q is taken as $q \equiv 1 \mod 2n$. In this paper we propose an optimization of a ring-LWE based encryption system from [6] in brief, which we now recall. The cryptosystem uses a global polynomial $a \in R_q$. The key generation, encryption and decryption operations are shown below.

1. $KeyGen(a)$: Two polynomials r_1 and $r_2 \in R_q$ are constructed from a discrete Gaussian distribution with small standard deviation. As such, the norms of r_1 and r_2 are small. The polynomial $p = r_1 - a \cdot r_2 \in R_q$ is computed. The public key is (a, p) and the private key is r_2.
2. $Enc(a, p, m)$: The message m is first encoded to $\bar{m} \in R_q$. Three polynomials e_1, e_2 and $e_3 \in R_q$ are sampled from the discrete Gaussian distribution. The ciphertext is composed of two polynomials $[c_1 = a \cdot e_1 + e_2;\ c_2 = p \cdot e_1 + e_3 + \bar{m} \in R_q]$.
3. $Dec(c_1, c_2, r_2)$: In the decryption phase, first a polynomial $m' = c_1 \cdot r_2 + c_2 \in R_q$ is computed. The original message m is recovered from m' using a decoder.

2.2 Polynomial Multiplication

There are many efficient algorithms in the literature to perform polynomial multiplication. A survey of fast multiplication algorithms can be found in [2]. In this paper, we use the Fast Fourier Transform (FFT) to implement polynomial multiplication in quasi-linear time complexity. For a detailed description of the FFT and its use in polynomial multiplication, readers may follow [3].

The Fast Fourier Transform : This special technique is an efficient way to compute the Discrete Fourier Transform (DFT) in quasi-linear time complexity. The n-point forward DFT of a polynomial $a(x) = \sum_{k=0}^{n-1} a_k x^k$ of degree $n-1$ consists of n evaluations of the polynomial $a(x)$ at n distinct points. When n is a power of two and the points are chosen as ω_n^k for $0 \leq k \leq n-1$, with ω_n the n-th primitive root of unity a very efficient algorithm can be derived. The forward DFT of $a(x)$ can be expressed as a polynomial $A(X) = \sum_{k=0}^{n-1} A_k X^k$ with the following coefficients.

$$A_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}, \quad k = 0 \ldots n-1 \tag{1}$$

The polynomial $a(x)$ can be recovered from the polynomial $A(x)$ using the inverse DFT which retrieves the coefficients as shown below.

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} A_j \omega_n^{-kj}, \quad k = 0 \ldots n-1 \tag{2}$$

Naive computation of the forward and backward DFTs have quadratic complexity. The FFT is a divide-and-conquer approach which takes advantage of the special properties of ω_n to compute the forward and backward DFTs in $\theta(n \log n)$ time. The input polynomial $a(x)$ of degree $n-1$ is divided into two smaller polynomials $a_{even}(x)$ and $a_{odd}(x)$ of degree $n/2-1$ as shown below.

$$\begin{aligned} a(x) &= a_{even}(x^2) + x a_{odd}(x^2) \\ a_{even}(x) &= a_0 + a_2 x + \cdots + a_{n-2} x^{n/2-1} \\ a_{odd}(x) &= a_1 + a_3 x + \cdots + a_{n-1} x^{n/2-1} \end{aligned} \tag{3}$$

The task of evaluating the polynomial $a(x)$ at the points ω_n^k for $0 \leq k \leq n-1$ reduces to evaluating the two smaller polynomials at the points ω_n^{2k}. However as $\omega_n^{2k} = \omega_{n/2}^k$, the evaluations are performed only at the $n/2$ distinct points $\omega_{n/2}^k$ for $0 \leq k \leq n/2-1$. A similar recursive approach is applied while evaluating the two smaller polynomials. After completion of the evaluations of $a_{even}(x)$ and $a_{odd}(x)$, the results are combined as per Eq. (3) to compute the polynomial $A(X)$ with the coefficients shown below.

$$A_k = a(\omega_n^k) = a_{even}(\omega_n^{2k}) + \omega_n^k a_{odd}(\omega_n^{2k}) \tag{4}$$

The factor ω_n^k used in the merging of the two smaller evaluations is called the *twiddle factor*. As the n-th primitive root of unity ω_n is a complex number, the traditional FFT algorithm involves floating point arithmetic which introduces computational inaccuracy. The Number Theoretic Transform (NTT) is an FFT defined over the ring $\mathbb{Z}_q$, and is free from inaccurate floating point arithmetic. In the NTT, ω_n is an n-th primitive root of unity modulo q and the coefficients in Eq. (1) and (2) are computed modulo q. In a ring R_q, the NTT exists if and only if n divides $d-1$ for every prime divisor d of q.

For the NTT computation, there are both recursive and iterative algorithms. The iterative version has advantage in terms of constant cost over the recursive one [3]. An iterative NTT is given in Algorithm 1 (source [3]). In line 2, a bit reverse operation is performed to rearrange the input coefficients of the input polynomial in a desired order. After the bit reverse operation, three nested loops perform the divide-and-conquer steps. In line 8, a multiplication by the twiddle factor is performed before the butterfly steps in line 10 and 11. The butterfly steps perform the operation in Eq. (4). All the computations in this iterative NTT algorithm are *in-place* and hence the algorithm is suitable for memory-constraint implementations.

Polynomial Multiplication : The multiplication of two polynomials $a(x)$ and $b(x)$ of degree $n-1$ is the polynomial $c(x) = a(x) \cdot b(x)$ of degree $2n-1$ which can be computed using NTT as shown below.

$$c(x) = NTT^{-1}_{\omega_{2n}}\big(NTT_{\omega_{2n}}(a) * NTT_{\omega_{2n}}(b)\big) \tag{5}$$

Here ω_{2n} is the $2n$-th primitive root of unity in R_q and $NTT_{\omega_{2n}}(\cdot)$ consists of $2n$ point-value pairs. The polynomial multiplication in Eq. (5) has complexity

Algorithm 1: *Iterative NTT*

Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \text{NTT}(a)$

```
1  begin
2      A ← BitReverse(a);
3      for m = 2 to n by m = 2m do
4          ω_m ← ω_n^{n/m} ;
5          ω ← 1 ;
6          for j = 0 to m/2 − 1 do
7              for k = 0 to n − 1 by m do
8                  t ← ωA[k + j + m/2] ;
9                  u ← A[k + j] ;
10                 A[k + j] ← u + t ;
11                 A[k + j + m/2] ← u − t ;
12             end
13             ω ← ω · ω_m ;
14         end
15     end
16 end
```

$\theta(n \log n)$ since both the forward and backward NTT computations have complexity $\theta(n \log n)$ and the component-wise multiplication has complexity $\theta(n)$.

When polynomial multiplication is performed in $R_q = Z_q[\mathbf{x}]/\langle f \rangle$, special property of the irreducible polynomial $f(x) = x^n + 1$ can be used to reduce the number of point-value pairs from $2n$ to n for both forward and backward NTT conversions. This special technique is known as the *negative wrapped convolution* which is explained below.

Let us consider the general case where the polynomial $c(x)$ of degree $2n-1$ is first computed as per Eq. (5) and then the reduction by the irreducible polynomial $x^n + 1$ is performed to produce the reduced polynomial $c^R(x) \in R_q$. If we look at the computation of the coefficients c_k and c_{k+n} of $c(x)$ using the backward NTT, then we find the following similarity between the two coefficients, where $0 \leq k \leq n-1$.

$$\begin{aligned} c_k &= \frac{1}{2n} \sum_{j=0}^{2n-1} C_j \omega_{2n}^{-kj} \\ c_{k+n} &= \frac{1}{2n} \sum_{j=0}^{2n-1} C_j \omega_{2n}^{-(k+n)j} = \frac{1}{2n} \sum_{j=0}^{2n-1} C_j \omega_{2n}^{-kj} (-1)^j \end{aligned} \tag{6}$$

After the reduction of $c(x)$ by the polynomial $x^n + 1$, the result $c^R(x)$ has its k-th coefficient c_k^R as shown below.

$$\begin{aligned} c_k^R = c_k - c_{n+k} &= \frac{1}{2n} \sum_{j=0}^{2n-1} C_j \omega_{2n}^{-kj} \left(1 - (-1)^j\right) \\ &= \frac{1}{n} \sum_{j=0}^{n-1} C_{2j+1} \omega_{2n}^{-k(2j+1)} \end{aligned} \tag{7}$$

From the above equation we see that the even-indexed coefficients C_{2j} (where $0 \leq j \leq n-1$) have no influence on the final result. Hence we do not need to compute A_{2j}, B_{2j} and C_{2j}. In the forward NTT, we evaluate $a(x)$ (and $b(x)$) only at the points ω_{2n}^{2j+1} as shown below.

$$A_{2k+1} = \sum_{j=0}^{n-1} a_j \omega_{2n}^{(2k+1)i} = \sum_{j=0}^{n-1} a_j (\psi \omega_n^k)^j \tag{8}$$

Here $\psi = \omega_{2n}$ is the square-root of ω_n and $0 \leq k \leq n-1$.

After the element wise multiplication we get $C_{2k+1} = A_{2k+1} \cdot B_{2k+1}$. We can compute the result $c^R(x)$ as per Eq. (7). However to keep similarity between the forward and backward NTT computation (for simplicity of hardware implementation), we first compute the coefficients of the polynomial $c'(x)$ as shown below.

$$c'_k = \sum_{j=0}^{n-1} C_{2j+1} \omega_n^{-kj} \tag{9}$$

Now the polynomial $c^R(x)$ is computed from $c'(x)$ using the following relation.

$$c^R(x) = \frac{1}{n} \sum_{j=0}^{n-1} c'_j \psi^{-j} x^j \tag{10}$$

When the modulus q is prime and the number of points n is a power of two, the negative wrapped convolution is possible iff $q \equiv 1 \mod 2n$.

3 Optimization in the NTT Computation

In this section we optimize the NTT and compare with the recent hardware implementations of polynomial multipliers [1, 11, 12]. First, the fixed cost involved in computing the powers of ω_n is reduced; then the pre-computation overhead in the forward negative-wrapped convolution is optimized; and finally an efficient memory access scheme is proposed which reduces the number of memory accesses during the NTT steps and also minimizes the number of block RAMs in the hardware architecture.

3.1 Optimization in the Fixed Computation Cost

In line 13 of Algorithm 1 the computation of the twiddle factor $\omega \leftarrow \omega \cdot \omega_m$ is performed in the j-loop. This computation can be considered as a fixed cost. However in [1, 11] the j-loop and the k-loop appears in an interchanged position. In this interchanged position, ω is updated in the innermost loop which is more frequent than in Algorithm 1. To avoid the computation of the twiddle factors, in [11] all the twiddle factors are kept ready in a pre-computed lookup

table (ROM) and are accessed whenever required. As the twiddle factors are not computed on-the-fly, the order of the two innermost loops does not result in an additional cost. However in [1] a more compact polynomial multiplier architecture is designed without using any lookup table and the twiddle factors are simply computed on-the-fly during the NTT computation. Hence in this implementation, the interchanged position of the two loops causes additional computational overhead. In this paper our target is to design a very compact polynomial multiplier. Hence we do not use any lookup table for the twiddle factors and follow Algorithm 1 to avoid extra computation.

3.2 Optimization in the Forward NTT Computation Cost

Here we revisit the forward negative-wrapped convolution technique used in [1, 11, 12]. For the two input polynomials $a(x)$ and $b(x)$, the polynomials $\bar{a}(x) = \sum_{j=0}^{n-1} a_j x^j \psi^j$ and $\bar{b}(x) = \sum_{j=0}^{n-1} b_j x^j \psi^j \in R_q$ are precomputed. The forward n-point NTT is performed on the polynomials $\bar{a}(x)$ and $\bar{b}(x)$. Here ψ is the square-root of ω_n modulo q.

We perform the forward part in a different way to reduce the precomputation overhead. In Eq. (8), we have described the computation of forward negative-wrapped convolution. The only difference between computation in Eq. (8) and the normal n-point NTT in Eq. (1) is the presence of ψ as a multiple. In the forward negative wrapped convolution, the input polynomial is evaluated in the n points $\psi\omega_n^k$ for $0 \leq k \leq n-1$. In this case Eq. (4) can be written as shown below.

$$A_k = a(\psi\omega_n^k) = a_{even}\big((\psi\omega_n^k)^2\big) + \psi\omega_n^k a_{odd}\big((\psi\omega_n^k)^2\big) \tag{11}$$

The final recursive splitting in both Eq. (4) and (11) generates two smaller polynomials each having only one coefficient. Hence the difference is only in the *combine* operations which involves the twiddle factors ω_n^k and $\psi\omega_n^k$ in Eq. (4) and (11) respectively. In the forward negative-wrapped convolution, we do a minor change in line 5 of Algorithm 1 and initialize the twiddle factor ω to ψ which is the square-root of ω_m. The modified forward NTT computation is shown in Algorithm 2.

3.3 Optimization in the Memory Access Scheme

During the NTT computation, the initial coefficients and the intermediate coefficients are stored in memory. When the number of coefficients is large, addressable memory such as RAM is more suitable for hardware implementation [1, 11]. However, in a RAM the number of memory locations that can be accessed in a cycle is limited by the number of read and write ports of the RAM. Thus the memory access could be a bottleneck in the NTT implementation. In this section we propose an efficient memory access scheme which optimizes the number of memory accesses, minimizes the number of block RAMs and still achieves maximum utilization of the arithmetic blocks present in the polynomial multiplier architecture.

Algorithm 2: *Iterative forward NTT*

Input: Polynomial $a(x) \in \mathbb{Z}_p[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_p$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_p[\mathbf{x}] = \text{NTT}(a)$

```
1  begin
2      A ← BitReverse(a);
3      for m = 2 to n by m = 2m do
4          ω_m ← ω_n^{n/m} ;
5          ω ← ψ = squareroot(ω_m) ;
6          for j = 0 to m/2 − 1 do
7              for k = 0 to n − 1 by m do
8                  t ← ωA[k + j + m/2] ;
9                  u ← A[k + j] ;
10                 A[k + j] ← u + t ;
11                 A[k + j + m/2] ← u − t ;
12             end
13             ω ← ω · ω_m ;
14         end
15     end
16 end
```

In the innermost loop (lines 8-to-11) of Algorithm 1, two coefficients $A[k+j]$ and $A[k+j+m/2]$ are first read from the memory and then arithmetic operations (one multiplication, one addition and one subtraction) are performed. At the end of the iteration, the new $A[k+j]$ and $A[k+j+m/2]$ are updated and are written back in memory. Thus during one iteration of the innermost loop, the arithmetic circuits are used only once, while the memory is read or written twice. This leads to idle states in the arithmetic blocks. To avoid under-utilization of the arithmetic blocks, the polynomial multiplier in [11] uses two parallel memory blocks for the two input polynomials. Parallel memory blocks help in providing continuous flow of required coefficients to the coefficient-multiplier circuit. In this paper, our target is to design a compact and efficient polynomial multiplier. We propose a memory access scheme which solves the under-utilization problem without using parallel memory blocks. This memory access scheme is based on the following observation.

Since the two coefficients $A[k+j]$ and $A[k+j+m/2]$ are processed together, the number of memory accesses can be minimized if these two coefficients are kept as a pair in a memory location. In that case the coefficient pair can be read from a memory location in one cycle, and then written back in the memory in one cycle after performing the arithmetic operations. Let us consider an example of 16-point NTT. During the first iteration of the m-loop in line 3 of Algorithm 1, the coefficient pairs $(A[0], A[1])$, $(A[2], A[3])$ etc. are processed in a sequence. Figure 1 shows a basic block-level diagram of the memory and the arithmetic circuits (one coefficient multiplier, one adder and one subtracter). Initially the memory (Figure 1) has the proper coefficient pairs in different locations. During the first iteration of the m-loop, for different values of k and $j = 0$, the coefficient pairs $(A[k+j], A[k+j+m/2])$ are first sequentially fetched from memory, then updated after the arithmetic operations and finally written back in the same memory location. In hardware, processing of the coefficient pairs can be

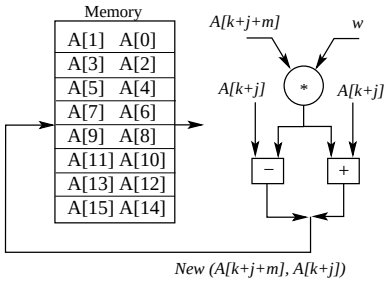

Fig. 1. Initial arrangement of the coefficients in memory in a 16-point NTT

performed in a pipeline. Hence the arithmetic components in Figure 1 are always used during the m-loop for $m = 2$, and thus resulting in no idle cycles.

However if the coefficient pairs are written back in their previous memory location, then the problem of not having the required coefficients in the same memory location reappears again in the next iteration for the incremented value of m. For example, when $m = 4$ in Algorithm 1, the coefficient pairs required for processing are $(A[0], A[2])$, $(A[4], A[6])$ etc. which are not present in the memory as pairs after completion of the m-loop for $m = 2$. The problem can be solved using the following observation.

- For any particular value of m in Algorithm 1, any coefficient is processed only once.
- The processed coefficients in the innermost loop in Algorithm 1 have a difference $m/2$ in their index for any particular value of m.
- For every increment in the m-loop in line 3 of Algorithm 1, the difference in the indexes of the coefficients in the pair doubles.

Let us consider two consecutive iterations $m = m_1$ and $m = m_2$ where $m_2 = 2m_1$ of the m-loop (line 3 in Algorithm 1). In the m_1-loop, for some j_1 and k_1 (maintaining the loop bounds in Algorithm 1) the coefficients $(A[k_1 + j_1], A[k_1 + j_1 + m_1/2])$ are processed as a pair. Then k increments to $k_1 + m_1$ and the processed coefficient pair is $(A[k_1 + m_1 + j_1], A[k_1 + m_1 + j_1 + m_1/2])$. Now from Algorithm 1 we see that the coefficient $A[k_1 + j_1]$ will be again processed in the m_2-loop with the coefficient $A[k_1 + j_1 + m_2/2]$. Since $m_2 = 2m_1$, the coefficient $A[k_1 + j_1 + m_2/2]$ is the coefficient $A[k_1 + j_1 + m_1]$ which is updated in the m_1-loop for $k = k_1 + m_1$. Hence during the m_1-loop if we swap the updated coefficients for $k = k_1$ and $k = k_1 + m_1$ and store $(A[k_1 + j_1], A[k_1 + j_1 + m_1])$ and $(A[k_1 + j_1 + m_1/2], A[k_1 + j_1 + 3m_1/2])$ as the coefficient pairs in the memory, then the coefficients in a pair have a difference of $m_2/2$ in their index and thus are ready for the m_2-loop. The operations during the two consecutive iterations $k = k_1$ and $k = k_1 + m_1$ during $m = m_1$ are now shown below. During the

Algorithm 3: *Iterative NTT : Memory Efficient*

Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \text{NTT}(a)$

begin
  $A \leftarrow BitReverse(a)$; /* Coefficients are stored in the memory as proper pairs */
  for $m = 2$ *to* $n/2$ *by* $m = 2m$ **do**
    $\omega_m \leftarrow m$-th primitiveroot(1) ;
    $\omega \leftarrow squareroot(\omega_m)$ or 1 /* Depending on forward or backward NTT */ ;
    for $j = 0$ *to* $m/2 - 1$ **do**
      for $k = 0$ *to* $n/2 - 1$ *by* m **do**
        $(t_1, u_1) \leftarrow (A[k + j + m/2], A[k + j])$ /* From MEMORY[k+j] */ ;
        $(t_2, u_2) \leftarrow (A[k + j + m], A[k + j + m/2])$ /* MEMORY[k+j+m/2] */ ;
        $t_1 \leftarrow \omega \cdot t_1$;
        $t_2 \leftarrow \omega \cdot t_2$;
        $(A[k + j + m/2], A[k + j]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
        $(A[k + m + j + m/2], A[k + m + j]) \leftarrow (u_2 - t_2, u_2 + t_2)$;
        $MEMORY[k + j] \leftarrow (A[k + j + m], A[k + j])$;
        $MEMORY[k + j + m/2] \leftarrow (A[k + j + 3m/2], A[k + j + m/2])$;
      end
      $\omega \leftarrow \omega \cdot \omega_n$;
    end
  end
  $m \leftarrow n$;
  $k \leftarrow 0$;
  $\omega \leftarrow squareroot(\omega_m)$ or 1 /* Depending on forward or backward NTT */ ;
  for $j = 0$ *to* $m/2 - 1$ **do**
    $(t_1, u_1) \leftarrow (A[j + m/2], A[j])$ /* From MEMORY[j] */ ;
    $t_1 \leftarrow \omega \cdot t_1$;
    $(A[j + m/2], A[j]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
    $MEMORY[j] \leftarrow (A[j + m/2], A[j])$;
    $\omega \leftarrow \omega \cdot \omega_m$;
  end
end

operations u_1, t_1, u_2 and t_2 are used as temporary storage registers.

$$(t_1, u_1) \leftarrow (A[k_1 + j_1 + m_1/2], A[k_1 + j_1]);$$
$$(t_2, u_2) \leftarrow (A[k_1 + m_1 + j_1 + m_1/2], A[k_1 + m_1 + j_1]);$$
$$t_1 \leftarrow \omega \cdot t_1;$$
$$t_2 \leftarrow \omega \cdot t_2;$$
$$(A[k_1 + j_1 + m_1/2], A[k_1 + j_1]) \leftarrow (u_1 - t_1, u_1 + t_1);$$
$$(A[k_1 + m_1 + j_1 + m_1/2], A[k_1 + m_1 + j_1]) \leftarrow (u_2 - t_2, u_2 + t_2);$$
$$MEMORY[k_1 + j_1] \leftarrow (A[k_1 + j_1 + m_1], A[k_1 + j_1]);$$
$$MEMORY[k_1 + j_1 + m_1/2] \leftarrow (A[k_1 + j_1 + 3m_1/2], A[k_1 + j_1 + m_1/2]);$$

The efficient memory access scheme is represented as Algorithm 3. In this algorithm for all values of $m < n$ except the last one $m = n$, two coefficient pairs are processed in the innermost loop and swap of the updated coefficients are performed before storing back in the memory. When $m = n$, no swap operations of the updated coefficients are required as this is the final iteration of the m-loop. As an example, Table 1 shows the memory contents during the execution of Algorithm 3 for $n = 16$. The column-heading represents (m, j, k) during the

iterations. The end loop in line 19 of Algorithm 3 for $m = 16$ performs no swap and is shown in the table using $\star$ symbol.

Initial	(2,0,0)	(2,0,6)	(4,0,0)	(4,0,4)	(4,1,4)	(8,3,0)	(16,7,0)$\star$
A_1 A_0	A_2 A_0	A_2 A_0	A_4 A_0	A_4 A_0	A_4 A_0	A_8 A_0	A_8 A_0
A_3 A_2	A_3 A_1	A_3 A_1			A_5 A_1	A_9 A_1	A_9 A_1
A_5 A_4		A_6 A_4	A_6 A_2	A_6 A_2	A_6 A_2	A_{10} A_2	A_{10} A_2
A_7 A_6		A_7 A_5			A_7 A_3	A_{11} A_3	A_{11} A_3
A_9 A_8		A_{10} A_8		A_{12} A_8	A_{12} A_8	A_{12} A_4	A_{12} A_4
A_{11} A_{10}		A_{11} A_9			A_{13} A_9	A_{13} A_5	A_{13} A_5
A_{13} A_{12}		A_{14} A_{12}		A_{14} A_{10}	A_{14} A_{10}	A_{14} A_6	A_{14} A_6
A_{15} A_{14}		A_{15} A_{13}			A_{15} A_{11}	A_{15} A_7	A_{15} A_7

Table 1. Memory content during the steps in a 16-point NTT

4 The Polynomial Multiplier Architecture

In this section we present a polynomial multiplier architecture that implements the optimization techniques described in Section 3. The polynomial multiplier architecture has been designed for the ring-LWE cryptosystem with dimension 256. The prime modulus used in the architecture is 1049089 which is also used in the ring-LWE256 implementation in [11]. The prime is a 21 bit number and thus the coefficients of the polynomials are also 21 bit numbers. Figure 2 shows the components present in the polynomial multiplier architecture.

During the forward NTT computation, the two input polynomials each having 256 coefficients are kept in memory; while only one polynomial (256 coefficients) is present in memory during the backward NTT computation. Hence if two coefficients are kept together in a memory word, then a total of 256 words are sufficient for the forward (and also backward) NTT. In Xilinx FPGAs, one 9K RAM slice can be configured as a memory of word size 36 bits and depth 256. In the case of ring-LWE256 with 21-bit prime modulus, the memory word-size should be 42 bits to store any two coefficients in one location. Since the RAM slices on Xilinx FPGAs can be upto 36 bits in width, the extra six bits can be stored on a LUT based RAM of width six. This helps in achieving better utilization of the FPGA resources.

The polynomial multiplier architecture in Figure 2 uses only one memory (RAM) block to store all the coefficients during the forward and backward NTT computations. The RAM in the figure is a hybrid of one block RAM (word size 36) and one LUT-based RAM (word size 6), each having a depth 256. During a forward NTT computation, the lower half of the RAM is used by the first polynomial, while the upper half of the RAM is used by the second polynomial.

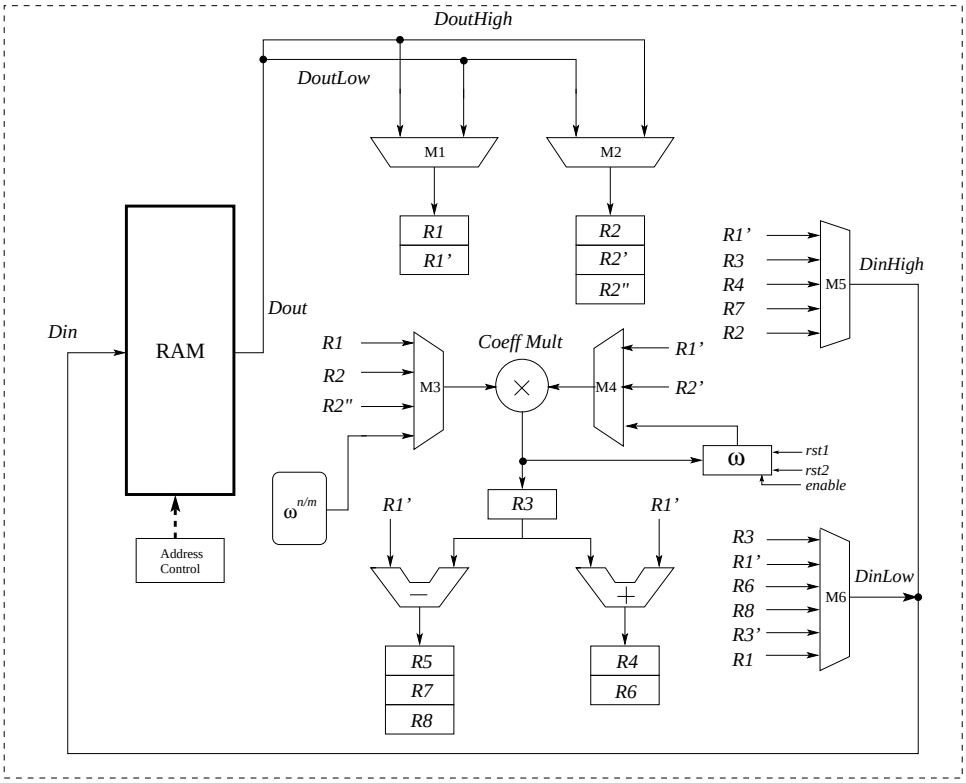

Fig. 2. Polynomial Multiplier Architecture

The registers present in the outputs of the multiplexers M1 and M2 are used to store the fetched coefficient pair. This special arrangement of the registers is used for the following purposes :

- To maintain a continuous flow of coefficients to the arithmetic components.
- To perform a re-arrangement of the coefficients after a forward NTT.

The registers R4 to R8 are used to store the updated coefficients and to perform the swap operations described in Section 3.3.

The coefficient multiplier uses DSP multipliers and also includes a modular reduction block. A register R3 (Figure 2) is used to break the critical path at the output of the coefficient multiplier. The modular adder and subtracter circuits perform the final addition and subtraction operations required to compute the updated coefficients. The multiplexers M5 and M6 are connected to different registers present in the architecture. Outputs from these two multiplexers are combined to form the input channel for the RAM block.

5 Experimental Results

We have evaluated the proposed polynomial multiplier architecture for ring-LWE256 with prime modulus 1049089. The results obtained from Xilinx ISE12.2 after place and route analysis are shown in Table 2. The coefficient-multiplier in the polynomial multiplier architecture requires two and four DSP slices respectively in Virtex-V and Spartan-VI FPGAs. The RAM in our architecture is composed of a block RAM of data-width 36 bits and a LUT-based distributed RAM of width six. The block RAM in Spartan VI is implemented using a 9K block RAM slice. However in Virtex V, the RAM consumes one 18K slice as the smallest size of block RAM in this family of FPGAs is 18K. The ISE tool reports few extra slices when the design is synthesized in Spartan VI.

Implementation	Device	Slices	Frequency (MHz)	RAM		Clock Cycles	Computation Time (μsec)
				Block-RAMs	LUT-slices		
Our	Virtex-V LX85	281	81	1 (18K)	19	4683	58
	Spartan-VI LX100	297	37	1 (9K)	19	4683	127
[11]	Spartan-VI LX100	640	218	5 (18K) + 1 (9K)	-	4806	22

Table 2. Performance of polynomial multipliers

The focus of our implementation is to reduce area and memory requirement without increasing the number of clock cycles. While the pipelined implementation in [11] targets to achieve speed. The polynomial multiplier in [1] uses a smaller prime modulus 65537 that simplifies the datapath of the architecture; hence comparison with this implementation is not fair. Our polynomial multiplier architecture achieves significant reduction in the number of block RAM slices compared to [11]. However, our architecture has lower operating frequency compared to the pipelined architecture in [11]. The reason behind lower frequency is the presence of a long critical path through the integer-multiplier and the modular reduction block that are present in the coefficient-multiplier. The focus of our implementation was to test the implementation possibilities of the optimization techniques proposed in this paper; so far, no additional effort has been spent on improving the critical path delay of the architecture. A pipeline strategy can be applied to improve the operating frequency of the polynomial multiplier at the cost of a marginal increase in the number of clock cycles. Moreover, in resource-constraint devices, the operating frequency is kept low in order to reduce power consumption. Our small-area-memory implementation with low clock cycle count can also be used for such platforms.

6 Optimization in the ring-LWE Cryptosystem

An actual hardware implementation of the ring-LWE public key cryptosystem is present in [12]. The implementation optimizes the number of NTT operations by using a scheme in which the fixed polynomials such as a, p and r_2 are kept

in forward-NTT transformed form. Thus during encryption or decryption operations, NTT computations for the fixed polynomials are not performed. With this optimization, the ring LWE public key cryptosystem in [12] requires a total of five NTT transforms.

In this paper we perform a further optimization in a ring-LWE based encryption system by reducing the number of NTT operations from five to *four*. Since the NTT is the most significant part in the LWE-encryption cryptosystem, the proposed optimization reduces computation cost by nearly 20%. The proposed ring-LWE public key scheme is described below.

1. $KeyGen(a)$: Two polynomials r_1 and $r_2 \in R_q$ are constructed from a discrete Gaussian distribution. The polynomial $p = r_1 - a \cdot r_2 \in R_q$ is computed. The NTT is performed on the three polynomials a, p and r_2 to generate $\tilde{a}$, $\tilde{p}$ and $\tilde{r_2}$. The public key is $(\tilde{a}, \tilde{p})$ and the private key is $\tilde{r_2}$ [12].
2. $Enc(\tilde{a}, \tilde{p}, m)$: The message is encoded and the three error polynomials are generated. The following computations are performed.

$$\begin{aligned} \tilde{e_1} &\leftarrow NTT(e_1) \\ \tilde{e_2} &\leftarrow NTT(e_2) \\ \tilde{c_1} &\leftarrow \tilde{a} * \tilde{e_1} + \tilde{e_2} \\ \tilde{c_2} &\leftarrow \tilde{p} * \tilde{e_1} + NTT(e_3 + \bar{m}) \end{aligned}$$

   The ciphertext is $(\tilde{c_1}, \tilde{c_2})$ and is transmitted to the decryption side.
3. $Dec(\tilde{c_1}, \tilde{c_2}, \tilde{r_2})$: After receiving the ciphertext, the polynomial m' is computed as $m' = INTT(\tilde{c_1} * \tilde{r_2} + \tilde{c_2}) \in R_q$. In the end decoding is performed on m' to recover the plaintext m.

The proposed ring-LWE public key scheme performs three NTT operations during encryption and one NTT during the decryption operation. The scheme requires both encryption and decryption to use a common primitive root of unity. It is also possible to perform the ring-LWE public key scheme in an alternative way : instead of computing $\tilde{c_2}$ during the encryption, we can compute $c_2 \leftarrow INTT(\tilde{p} * \tilde{e_1}) + e_3 + \bar{m}$ and can decrypt as $m' = INTT(\tilde{c_1} * \tilde{r_2}) + c_2$. However, in this case, the cost is higher since an inverse NTT requires an extra scaling compared to the forward NTT.

7 Conclusion and Future Work

This paper proposed two layers of optimization in implementing a ring-LWE based encryption system. On the top layer, the number of NTT operations required is reduced from five to four thus resulting in a speed-up of nearly 20%. On the lower layer, the polynomial multiplication is improved by (1) reducing the fixed computation cost and (2) implementing an efficient memory access scheme

that increases the utilization of the arithmetic components. The proposed optimization techniques are realized in a polynomial multiplier architecture for the ring-LWE public key cryptosystem of dimension 256.

The improvements proposed in this paper are mainly on the computation level, whereas the architecture level has received limited focus in this work. The polynomial multiplier has a low operating frequency due to the long critical paths in the architecture. Though our implementation satisfies the requirements of a resource-constraint platform, we plan to improve the polynomial multiplier on the *hardware-architecture* level to match timing requirements of high-speed applications. A pipeline strategy is one key idea which could reduce the delay of the architecture significantly. At the same time, our work enhances the scopes for parallelism in designing a very fast polynomial multiplier architecture by reducing the computation cost, by minimizing the bottlenecks in memory access, and by utilizing the arithmetic blocks more efficiently.

Acknowledgment

This work was supported in part by the Research Council KU Leuven: TENSE (GOA/11/007), by iMinds, by the Flemish Government, FWO G.0550.12N and by the Hercules Foundation AKUL/11/19.

References

1. A. Aysu, C. Patterson, and P. Schaumont. Low-cost and Area-efficient FPGA Implementations of Lattice-based Cryptography. In *HOST*, pages 81–86. IEEE, 2013.
2. D. Bernstein. Fast Multiplication and its Applications. *Algorithmic Number Theory*, 44:325–384, 2008.
3. T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms.* http://staff.ustc.edu.cn/∼csli/graduate/algorithms/book6/toc.htm.
4. T. Frederiksen. A Practical Implementation of Regev's LWE-based Cryptosystem. In *http://daimi.au.dk/ jot2re/lwe/resources/*, 2010.
5. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes. In *Cryptographic Hardware and Embedded Systems CHES 2012*, volume 7428 of *LNCS*, pages 512–529. Springer Berlin, 2012.
6. R. Lindner and C. Peikert. Better Key Sizes (and Attacks) for LWE-based Encryption. *CT-RSA 2011*, pages 319–339, 2011.
7. V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 2010.
8. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 2001.
9. D. Micciancio. *Lattices in Cryptography and Cryptanalysis.* 2002.
10. P. Q. Nguyen and J. Stern. The Two Faces of Lattices in Cryptology. In *Cryptography and Lattices, International Conference (CaLC 2001)*, volume 2146 of *LNCS*, pages 146–180. Springer-Verlag, Berlin, 2001.

11. T. Pöppelmann and T. Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In A. Hevia and G. Neven, editors, *Progress in Cryptology LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158. Springer Berlin, 2012.
12. T. Pöppelmann and T. Güneysu. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Selected Areas in Cryptography SAC 2013*, LNCS. Springer-Verlag, Burnaby, Canada, 2013, Preprint.
13. O. Regev. Quantum Computation and Lattice Problems. *SIAM J. Comput.*, 33(3):738–760, Mar. 2004.
14. O. Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
15. O. Regev. Lattice-Based Cryptography. In C. Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, volume 4117 of *LNCS*, pages 131–141. Springer Berlin, 2006.