Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture

Ahmet Can Mert[1], Erdinc Ozturk[1], and Erkay Savas[1]

Sabanci University
Orta Mahalle, 34956 Tuzla, Istanbul, Turkey
{acmert,erdinco,erkays}@sabanciuniv.edu
https://www.sabanciuniv.edu/en/

Abstract. In this paper, we present an optimized FPGA implementation of a novel, fast and highly parallelized NTT-based polynomial multiplier architecture, which proves to be effective as an accelerator for lattice-based homomorphic cryptographic schemes. As input-output (I/O) operations are as time-consuming as NTT operations during homomorphic computations in a host processor/accelerator setting, instead of achieving the fastest NTT implementation possible on the target FPGA, we focus on a balanced time performance between the NTT and I/O operations. Even with this goal, we achieved the fastest NTT implementation in literature, to the best of our knowledge. For proof of concept, we utilize our architecture in a framework for Fan-Vercauteren (FV) homomorphic encryption scheme, utilizing a hardware/software co-design approach, in which NTT operations are offloaded to the accelerator while the rest of operations in the FV scheme are executed in software running on an off-the-shelf desktop computer. Specifically, our framework is optimized to accelerate Simple Encrypted Arithmetic Library (SEAL), developed by the Cryptography Research Group at Microsoft Research [27], for the FV encryption scheme, where forward and inverse NTT operations are utilized extensively for large degree polynomial multiplications. The hardware part of the proposed framework targets XILINX VIRTEX-7 FPGA device, which communicates with its software part via a PCIe connection. Offloading forward/inverse NTT and coefficient multiplication operations on FPGA, taking into account the time expended on I/O operations, the proposed framework achieves almost x11 latency speedup for the offloaded operations compared to their pure software implementations. With careful pipelining, overlapping I/O operations with actual polynomial multiplication computations, and assuming one of the operands for the polynomial multiplication operation is already inside the FPGA (valid assumption for encrypt/decrypt operations for homomorphic applications), we achieved a throughput of almost 800k polynomial multiplications per second, for polynomials of degree 1024 with 32-bit coefficients.

Keywords: Number Theoretic Transform, Large-Degree Polynomial Multiplication, Fan-Vercauteren, SEAL, FPGA

1 Introduction

Fully Homomorphic Encryption (FHE) allows computations on encrypted data eliminating the need for the access to plaintext data. FHE schemes provide privacy in various applications, such as privacy-preserving processing of sensitive data in cloud computing; albeit this being theoretically possible except for those requiring relatively simple homomorphic operations. The idea of FHE was first introduced in 1978 [25] and it had been an open problem until Gentry constructed the first functioning FHE scheme in 2009 [12], [13]. Since the introduction of Gentry's scheme, various practical FHE schemes have been introduced [17], [11], [28]. Despite the tremendous performance improvement of FHE schemes over the years, homomorphic computation is not yet quite feasible for many cloud applications. There is still ongoing research and race to improve the performance of arithmetic building blocks of the working FHE schemes. Different implementations and constructions were developed to introduce practical hardware and software implementations of FHE schemes, such as [26], [8], HElib [14], NFLlib [1], cuHe [9]. With the motivation of achieving a practical FHE implementation, we focus on improving performance of the most time consuming arithmetic building block of many FHE schemes in literature: large degree polynomial multiplication. For proof of concept, we aim to obtain a framework to accelerate the Fan-Vercauteren (FV) encryption scheme for homomorphic operations [11].

There are various software and hardware implementations of the FV scheme in the literature. Cryptography Research Group at Microsoft Research developed Simple Encrypted Arithmetic Library (SEAL) [27], providing a simple and practical software infrastructure using the FV homomorphic encryption scheme for homomorphic applications [11]. SEAL already gained recognition in the literature [18], [29], [4], [7], [2]. The work in [18] performs private comparison of two input integers using SEAL. In [29], the authors propose a privacy-preserving recommendation service and utilize the SEAL library for homomorphic operations. The GPU implementation in [4] is compared with the SEAL performance. The SEAL team recently announced highly efficient SealPIR, which is a Private Information Retrieval tool that allows a client to download an element from a database stored by a server without revealing which element is downloaded [2]. In our proof-of-concept framework, we utilize our NTT-based polynomial multiplier design to accelerate SEAL software by offloading large degree polynomial multiplication operations to the hardware accelerator implemented on a target FPGA device.

Utilizing FPGA architectures as accelerator to software implementations is currently a popular topic. Microsoft Research initiated Project Catapult in 2010 with the aim of investigating FPGA architectures as an alternative compute layer in their cloud computing infrastructure [23]. This project was very successful and they report a 95% increase in ranking throughput of each server that has an FPGA connected to the CPU.

In general, implementations of specialized hardware architectures for specific operations provide significant speed-up over software implementations. On the other hand, it is neither effective nor practical to offload all sorts of computation to hardware accelerators. Similarly, FHE schemes require highly diverse set of involved operations, many of which can be efficiently implemented in software and do not take up much time while quite a high percentage of their execution time is expended in only few operations. In most FHE schemes, the most-time consuming operation is large degree polynomial multiplication that involves vast number of modular multiplication operations over integers, majority of which is highly parallelizable. Nevertheless, software performance is bounded by the number of integer multipliers existing in CPU architectures as most modern CPU architectures feature a single 64-bit integer multiplier per core, limiting the level of parallelization. Therefore, it makes perfect sense to execute those operations in a hardware accelerator, which should be designed to improve the overall performance of software implementation of a FHE scheme by taking advantage of parallelizable operations and carefully designed, highly optimized hardware functional units.

Finally, offloading computation to an accelerator results in overhead due to the time spent in the network stack in both ends of the communication and actual transfer of data, which we refer as the I/O time. As accelerating FHE involves handling of large degree polynomials, this overhead can be prohibitively high if the nature and the cost of the offloading are not factored in the accelerator design.

To this end, two crucial design goals are considered in this work: i) hardware accelerator architecture should be designed to provide significant levels of speedup over software implementations and ii) the overhead due to communication (I/0 time) between hardware and software components of the framework should be taken into account as a design parameter or constraint. Most works in the literature focus solely on the first goal and report no accurate speedup values subsuming the I/O time. In this paper, we aim to address this problematic by providing a fully working prototype of a framework consisting of an FPGA implementing a highly efficient NTT accelerator and SEAL library running on a CPU.

Our Contribution Our contributions in this paper are listed as follows:

- We present a novel FPGA implementation of a fast and highly parallelized NTT-based polynomial multiplier architecture. We introduce several optimizations for the NTT operations. For efficient modular arithmetic, we employ lazy reduction techniques as explained in [30]. We also slightly modify the NTT operation loops in order to be able to efficiently parallelize NTT computations. Since input-output (I/O) operations are as important as NTT operations running on the FPGA, instead of achieving the fastest NTT implementation possible on the target FPGA, we focus on a balanced performance between the NTT and I/O operations on the FPGA. Also, since our implementations are targeting crytptographic applications, for security NTT hardware is designed to run in constant time for every possible input combination.

- We propose a framework including a high performance FPGA device, which is connected to a host CPU. Our proposed framework interfaces the CPU and the FPGA via a fast PCIe connection, achieving a $\tilde{3}$2Gbps half-duplex I/O speed. For proof of concept, we accelerate Number Theoretic Transform (NTT) operations that are heavily utilized in SEAL for large degree polynomial multiplications. Every time an NTT function is invoked by SEAL, the computation is offloaded to the FPGA device via the fast PCIe connection. Our design utilizes 1024-degree polynomials to achieve 128-bit security level. With our approach, latency of NTT operation is improved by almost 11x with about a 17% utilization of the VIRTEX-7 resources[1]. With careful pipelining, I/O operations can be overlapped with actual NTT computations on hardware and additional 3x throughput performance can be achieved for pure NTT operations.
- We introduce a novel modular multiplier architecture for any NTT-friendly prime modulus, which provides comparable time performance to those using special primes.
- As the accelerator framework provides a simple interface and supports a range of modulus lengths for polynomial coefficients it can easily be configured for use with other FHE libraries relying on ring learning with errors (ring LWE) security assumption.

2 Background

In this section we give definition of the FV scheme as presented in [11] and arithmetic operations utilized in this scheme.

2.1 FV Homomorphic Encryption Scheme

In [11], the authors present an encryption scheme based on Ring Learning with Errors (RLWE) problem [19]. The RLWE problem is simply a ring based version of the LWE problem [24] and is formulated as follows in [19].

Definition 1. *Decision-RLWE: For security parameter λ, let $f(x)$ be a cyclotomic polynomial $\Phi_m(x)$ with $deg(f) = \varphi(m)$ depending on λ and set $R = \mathbb{Z}[x]/(f(x))$. Let $q = q(\lambda) \geq 2$ be an integer. For a random element $s \in R_q$ and a distribution $\chi = \chi(\lambda)$ over R, denote with $A_{s,\chi}^{(q)}$, the distribution obtained by choosing a uniformly random element $\mathbf{a} \leftarrow R_q$ and a noise term $\mathbf{e} \leftarrow \chi$ and outputting $(\mathbf{a}, [\mathbf{a} \cdot \mathbf{s} + \mathbf{e}]_q)$. The $Decision-RLWE_{d,q,\chi}$ problem is to distinguish between the distribution $A_{s,\chi}^{(q)}$ and the uniform distribution $U(R_q^2)$.*

The above decision problem leads to the following encryption scheme as described in [19]. Let the plaintext and ciphertext spaces taken as R_t and R_q, respectively, for some integer $t > 1$. Let $\Delta = \lfloor q/t \rfloor$ and denote with $r_t(q) = q$

[1] Although 22% of the BRAM primitives are utilized, most of them are used for precomputed constants.

mod t. Then we have $q = \Delta \cdot t + r_t(q)$. We remark that q nor t have to be prime, nor that t and q have to be coprime. Let $\lfloor . \rceil$ and $[.]_q$ represent round to nearest integer and the reduction by modulo q operations, respectively. Let $\mathbf{a} \xleftarrow{\$} \mathbf{S}$ represents that $\mathbf{a}$ is uniformly sampled from the set $\mathbf{S}$. Secret key generation, public key generation, encryption and decryption operations described in Textbook-FV are shown below.

- **SecretKeyGen**: $s \xleftarrow{\$} R_2$.
- **PublicKeyGen**: $a \xleftarrow{\$} R_q$ and $e \leftarrow \chi$.
$$(p_0, p_1) = ([-(a \cdot s + e)]_q, a)$$
- **Encryption**: $m \in R_t$, $u \xleftarrow{\$} R_2$ and $e_1, e_2 \leftarrow \chi$.
$$(c_0, c_1) = ([\Delta \cdot m + p_0 \cdot u + e_1]_q, [p_1 \cdot u + e_2]_q)$$
- **Decryption**: $m \in R_t$
$$m = [\lfloor \tfrac{t}{q} [c_0 + c_1 \cdot s]_q \rceil]_t$$

2.2 Number Theoretic Transform

One of the high level fundamental operations in the FV scheme is the multiplication of two polynomials of very large degrees. Recently, there has been many publications in literature about multiplication of two large degree polynomials and the NTT-based multiplication schemes which provide the most suitable algorithms for efficient multiplication of large degree polynomials. In this work, we utilize the iterative NTT scheme shown in Algorithm 1 as shown in [26].

2.3 Modular Arithmetic

NTT arithmetic involves a large amount of modular addition, subtraction and multiplication operations. For efficient modular arithmetic operations, we employ techniques discussed in [30]. In this section we present hardware-friendly constant-time modular arithmetic algorithms. For the rest of the section, we assume a K-bit modulus M. Our modular arithmetic operations compute numbers in the range $[0, 2^K - 1]$, instead of $[0, M - 1]$.

Modular Addition A hardware-friendly constant-time partial modular addition operation is shown in Algorithm 2. Assume largest values for A and B are $2^K - 1$, and smallest value for M is $2^{K-1} + 1$.

$$A_{max} + B_{max} = (2^K - 1) \cdot 2 = 2^{K+1} - 2 \tag{1}$$

$$A_{max} + B_{max} - M_{min} = (2^K - 1) \cdot 2 - (2^{K-1} + 1) = 3 \cdot 2^{K-1} - 3 \tag{2}$$

```
Input: Polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$, primitive $n$-th root of unity $\omega \in \mathbb{Z}_q, q \equiv 1 \ mod \ 2n$
Output: Polynomial $a(x) =$NTT$(a) \in \mathbb{Z}_q[x]$
 1: $a \leftarrow$BitReverse$(a)$
 2: for $i$ from 2 by $2i$ to $n$ do
 3:     $\omega_i \leftarrow \omega_n^{n/i}$, $\omega \leftarrow 1$
 4:     for $j$ from 0 by 1 to $i/2-1$ do
 5:         for $k$ from 0 by $i$ to $n-1$ do
 6:             $U \leftarrow a[k+j]$
 7:             $V \leftarrow \omega \cdot a[k+j+i/2]$
 8:             $a[k+j] \leftarrow U+V$
 9:             $a[k+j+i/2] \leftarrow U-V$
10:         end for
11:         $\omega \leftarrow \omega \cdot \omega_i$
12:     end for
13: end for
14: return $a$
```

Algorithm 1: Iterative Number Theoretic Transform

$$A_{max} + B_{max} - 2 \cdot M_{min} = (2^K - 1) \cdot 2 - (2^{K-1} + 1) \cdot 2 = 2^K - 4 \qquad (3)$$

Results of equation 1 and equation 2 are $K+1$-bit numbers, and result of equation 3 is a K-bit number. This shows that after an addition operation, at most 2 subtraction operations are required to reduce the result of the addition operation back to K bits. Therefore, in Algorithm 2, the result C is guaranteed to be a K-bit number. As can be seen, Algorithm 2 is built to be a constant-time operation.

```
Input: $A, B, M$ (K–bit positive integers)
Output: $C \equiv A + B \ mod \ M$ (K–bit positive integer)
 1: $T1 = A + B$
 2: $T2 = T1 - M$
 3: $T3 = T1 - 2 \cdot M$
 4: if $(T2 < 0)$ then
 5:     $C = T1$
 6: else if $(T3 < 0)$ then
 7:     $C = T2$
 8: else
 9:     $C = T3$
10: end if
```

Algorithm 2: Modular Addition Algorithm

Modular Subtraction For efficiency, we use partial modular subtraction operations, instead of full modular subtraction. Our algorithm is shown in Algorithm 3. Assume A is 0, largest value for B is $2^K - 1$, and smallest value for M is $2^{K-1} + 1$.

$$A_{min} - B_{max} = 0 - (2^K - 1) = -2^K + 1 \tag{4}$$

$$A_{min} - B_{max} + M_{min} = -2^K + 1 + (2^{K-1} + 1) = -2^{K-1} + 2 \tag{5}$$

$$A_{min} - B_{max} + 2 \cdot M_{min} = -2^K + 1 + 2 \cdot (2^{K-1} + 1) = 3 \tag{6}$$

Results of equation 4 and equation 5 are negative numbers, and result of equation 6 is a K-bit positive number. This shows that after a subtraction operation, at most 2 addition operations are required to guaranteed a positive result. Therefore, in Algorithm 3, the result C is guaranteed to be a positive K-bit number. As can be seen, Algorithm 3 is built to be a constant-time operation.

```
Input: $A, B, M$ (K-bit positive integers)
Output: $C \equiv A - B \ mod \ M$
 1: $T1 = A - B$
 2: $T2 = T1 + M$
 3: $T3 = T1 + 2 \cdot M$
 4: if ($T2 < 0$) then
 5:     $C = T3$
 6: else if ($T1 < 0$) then
 7:     $C = T2$
 8: else
 9:     $C = T1$
10: end if
```

Algorithm 3: Modular Subtraction Algorithm

Modular Multiplication For our entire hardware, we utilized Montgomery Reduction algorithm [20], for reasons explained in Section 3. Word-level version of the Montgomery Reduction Algorithm is shown in Algorithm 4. As can be seen from the algorithm, Montgomery Reduction is a constant-time operation.

3 NTT Multiplier Architecture

In this section, design techniques used for an efficient and scalable NTT multiplier is explained. For proof of concept design, we chose to implement a 1024-degree polynomial multiplication architecture targeting a 128-bit security level. For the rest of the paper, n denotes the degree of the polynomial, q denotes the prime used as modulus. Instead of fixing the modulus size and the modulus, we implemented a scalable architecture supporting modulus lenghts between 22 and 32 bits. Our techniques can easily be extended and optimized for fixed-length moduli. Our modular multiplier works for all NTT-friendly primes with the property $q \equiv 1 \ mod \ 2n$, as shown in Algorithm 4.

```
Input: $C = A \cdot B$ (a $2K$-bit positive integer)
Input: $M$ (a $K$-bit positive integer)
Input: $mu$ ($w$-bit positive integer $mu = -M^{-1} mod 2^w$, $w <= K$)
Output: $Res = C \cdot R^{-1}\ mod\ M$ where $R = 2^{K+w}\ mod\ M$
 1: $L = \lceil K/w \rceil$
 2: $T1 = C$
 3: for $i$ from 0 to $L-1$ do
 4:     $T2 = T1\ mod\ 2^w$
 5:     $T3 = (T2 \cdot mu)\ mod\ 2^w$
 6:     $T1 = \lfloor (T1 + (T3 \cdot M))/2^w \rfloor$
 7: end for
 8: $T4 = T1 - M$
 9: if ($T4 < 0$) then
10:     $Res = T1$
11: else
12:     $Res = T4$
13: end if
```

Algorithm 4: Word-Level Montgomery Reduction Algorithm

3.1 Core Multiplier

To optimize large polynomial multiplications for FV scheme, a fast and efficient modular multiplier needs to be designed and utilized. In this work, we designed a Modular Multiplier utilizing Montgomery Reduction techniques with a lazy reduction approach as explained in [30]. Our Modular Multiplier Architecture is optimized for modulus lengths between 22 and 32 bits.

We designed a 32-bit multiplier as shown in Figure 1. Since we are targeting and FPGA architecture, we used 16-bit core multipliers, because of DSP size limitations. On Spartan-6 Architectures, DSP slices include 18-bit signed multipliers and on Virtex-7 Architectures, DSP slices include 18x25-bit signed multipliers. To follow literature, we chose to implement our multiplier for both architectures, therefore we picked a core multiplier length of 16 bits.

Our NTT architecture is fully pipelined, therefore pipeline registers shown in 1 does not affect the throughput of the overall architecture in terms of clock cycles, improving the overall performance in terms of seconds significantly.

3.2 Modular Reduction

After a 32-bit multiplication operation, the result needs to be reduced back to the bit-length of the modulus. For a scalable architecture, we modified Algorithm 4 to achieve a fast and efficient modular reduction operation. For efficiency, we utilize the property:

$$q \ \equiv\ 1\ mod\ 2n \tag{7}$$

Any NTT-friendly prime q with this property can be written as:

$$q \ = qH \cdot 2^{log_2 2n} + 1 \tag{8}$$

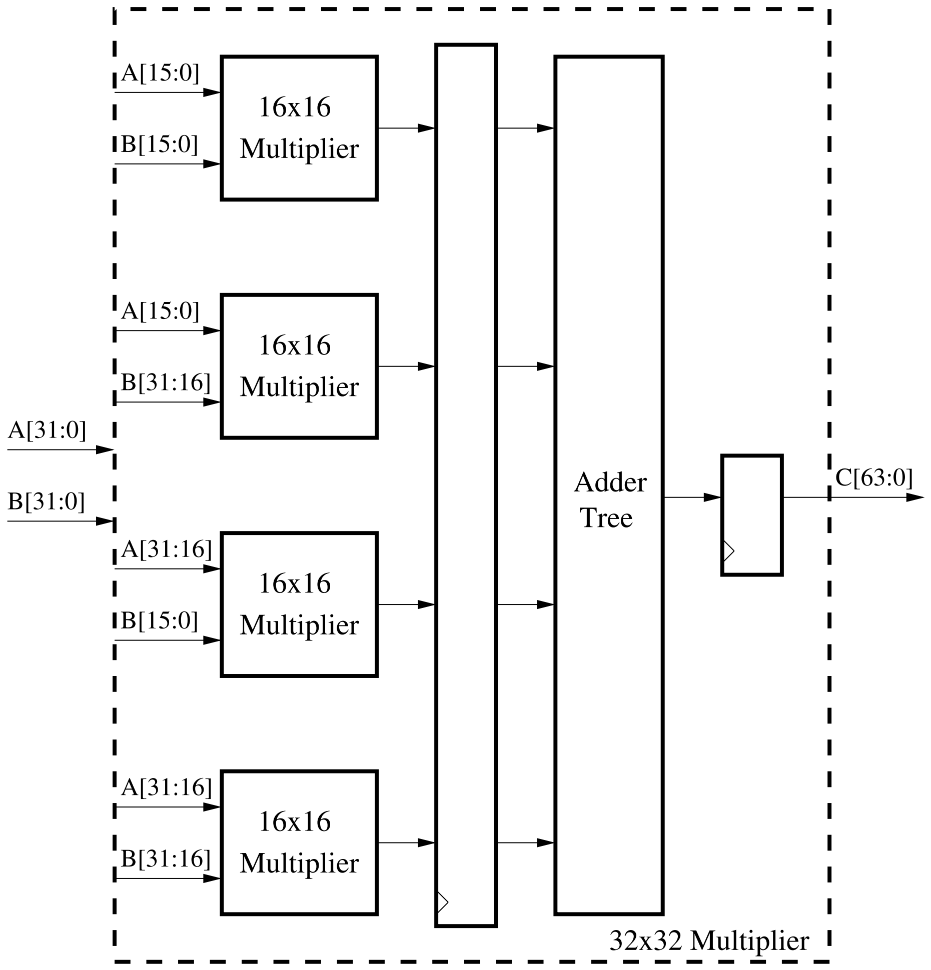

Fig. 1. 32-bit Multiplier

For our proof of concept design, $n = 1024$ and $log_2 2n = 11$, which yields:

$$q \ = qH \cdot 2^{11} + 1 \tag{9}$$

For Montgomery Reduction operation, if we select word size $w = 11$,

$$mu = -q^{-1} mod\ 2^{11} \equiv -1\ mod\ 2^{11} \tag{10}$$

Utilizing this property, we rewrite Montgomery Reduction as shown in Algorithm 5.

```
Input: $C = A \cdot B$ (a $2K$-bit positive integer, $44 \leq K \leq 64$)
Input: $q$ (a $K$-bit positive integer, $q \ = qH \cdot 2^{11} + 1$)
Output: $Res = C \cdot R^{-1}\ mod\ M$ where $R = 2^{33}\ mod\ M$
 1: $T1 = C$
 2: for $i$ from 0 to 2 do
 3:     $T1H = T1 >> 11$
 4:     $T1L = T1\ mod\ 2^{11}$
 5:     $T2 = 2's\ complement\ of\ T1L$
 6:     $carry = T2[10]\ OR\ T1L[10]$
 7:     $T1 = T1H + (qH \cdot T2) + carry$
 8: end for
 9: $T4 = T1 - q$
10: if $(T4 < 0)$ then
11:     $Res = T1$
12: else
13:     $Res = T4$
14: end if
```

Algorithm 5: Word-Level Montgomery Reduction algorithm modified for NTT-friendly primes

Flow of operations for Algorithm 5 is shown in Figure 2.

To guarantee that one subtraction at the end of Algorithm 5 is enough, $K < (3 * 11)$ needs to be satisfied. For $K < (2 * 11)$, 2 iterations are required instead of 3. Our algorithm can easily be modified to scale for other n. For example, for $n = 2048$, $w = 12$ and for a modulus of length $(4*12) < K < (5*12)$, 5 iterations are required. For $n = 4096$, $w = 13$ and for a modulus of length $(4 * 13) < K < (5 * 13)$, 5 iterations are required.

Hardware design for Algorithm 5 is shown in Figure 3. Calculation of carryin1, carryin2 and carryin3 signals is not shown in the Figure, it can easily be extracted from Algorithm 5, Step 6. $XY + Z$ is a multiply-accumulate operation, which can be realized using DSP blocks inside the FPGA. Each DSP slice has an optional output register, which can be utilized as the pipeline register, eliminating the need to utilize FPGA fabric registers for pipelining.

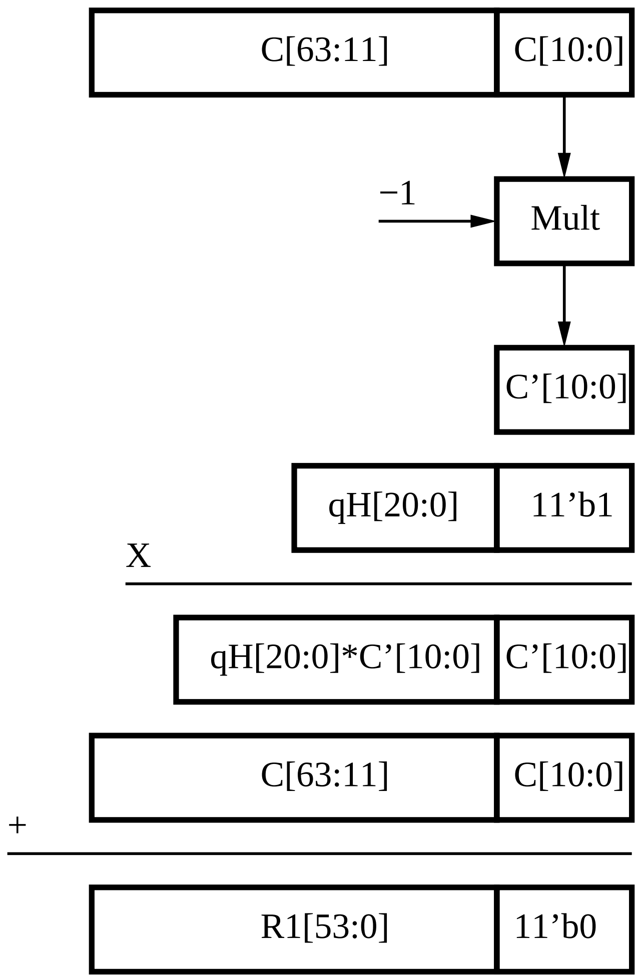

Fig. 2. Flow of operations for Word-Level Montgomery Reduction algorithm modified for NTT-friendly primes

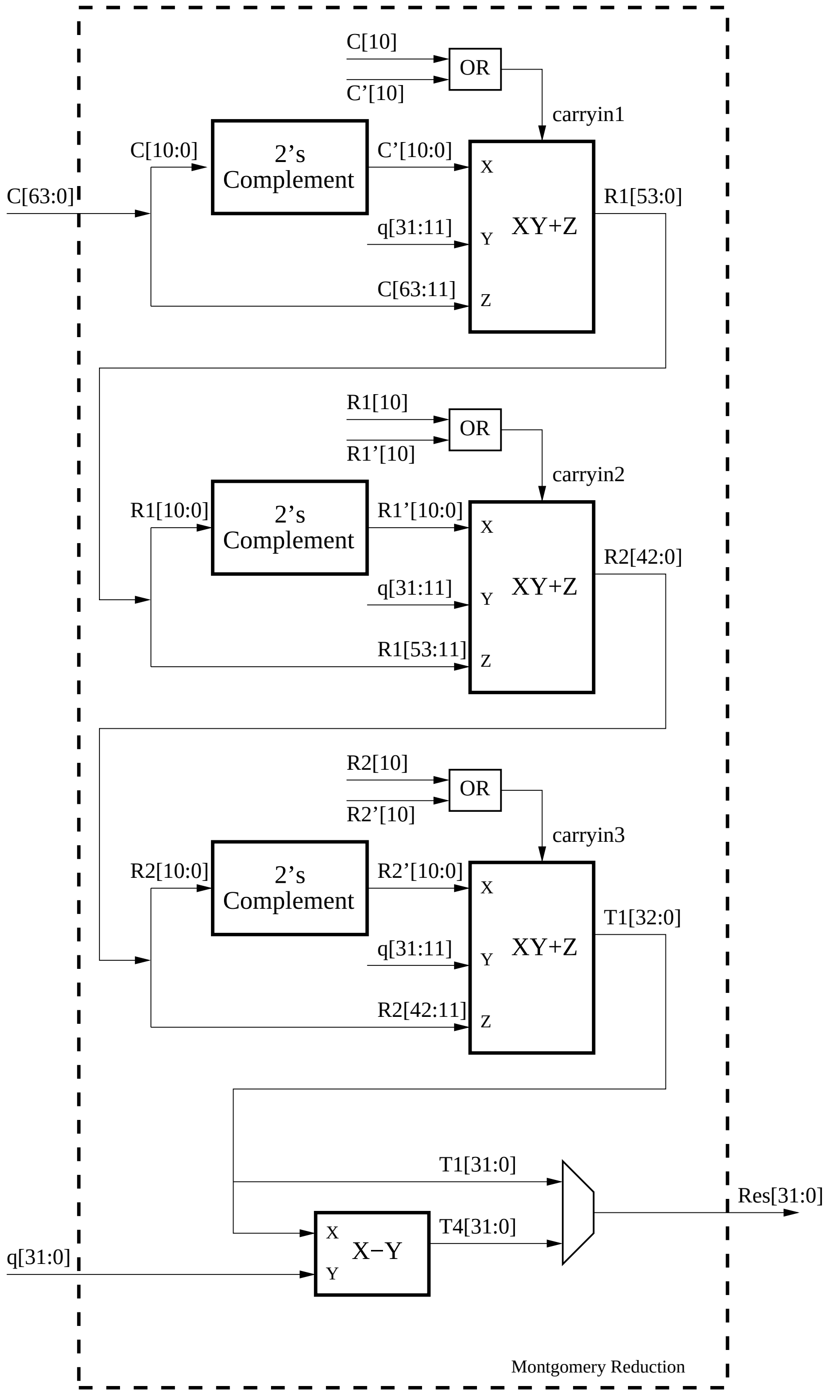

Fig. 3. Word-Level Montgomery Reduction algorithm modified for NTT-friendly primes

3.3 NTT Unit

To achieve optimized performance for NTT computations, we modified the iterative NTT algorithm that was shown in Algorithm 1. Our optimizations are based on modifications shown in [16]. Iterative NTT algorithm that we utilized is shown in Algorithm 6.

```
Input: Polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$, primitive $n$-th root of unity $\omega \in \mathbb{Z}_q$,
    primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q, N = 2^n$
Output: Polynomial $a(x) =$NTT$(a) \in \mathbb{Z}_q[x]$
 1: $curr\_\psi = 1$
 2: for $i$ from 0 by 1 to $N-1$ do
 3:     $a[i] = a[i] \cdot curr\_\psi$
 4:     $curr\_\psi = curr\_\psi \cdot \psi$
 5: end for
 6: for $i$ from 1 by 1 to $n$ do
 7:     $m = 2^{n-i}$
 8:     for $j$ from 0 by 1 to $2^{i-1} - 1$ do
 9:         $t = 2 \cdot j \cdot m$
10:         for $k$ from 0 by 1 to $m-1$ do
11:             $curr\_\omega = \omega[2^{i-1}k]$
12:             $U \leftarrow a[t+k]$
13:             $V \leftarrow a[t+k+m]$
14:             $a[t+k] \leftarrow U + V$
15:             $a[t+k+m] \leftarrow \omega \cdot (U - V)$
16:         end for
17:         $\omega \leftarrow \omega \cdot \omega_i$
18:     end for
19: end for
20: return $a$
```

Algorithm 6: Modified Iterative Number Theoretic Transform

It should be noted that this NTT operation is not a complete NTT operation. The resulting polynomial coefficients are not in correct order. We need to do a permutation operation in order to be able to get a correct NTT result. However, since we are in NTT domain and every operand that in the NTT domain will have the same scrambled order, we can leave the result of this operation as it is without doing the permutation. For polynomial multiplication, two polynomials will be converted to NTT domain and their inner multiplication will be computed. This operation will yield a result that is still in the same scrambled order.

After inner multiplication of operands in NTT form, we apply inverse NTT operation to bring the operand back to its polynomial domain. With slight modifications to inverse NTT operations, we were able to reverse this scrambling of NTT operands without any extra permutation operations, which yielded a lower latency for NTT operations.

In order to realize the most inner loop of the nested for loops shown in Algorithm 6, we designed an NTT unit, shown in Figure 4. Latency of this NTT unit is 5 clock cycles.

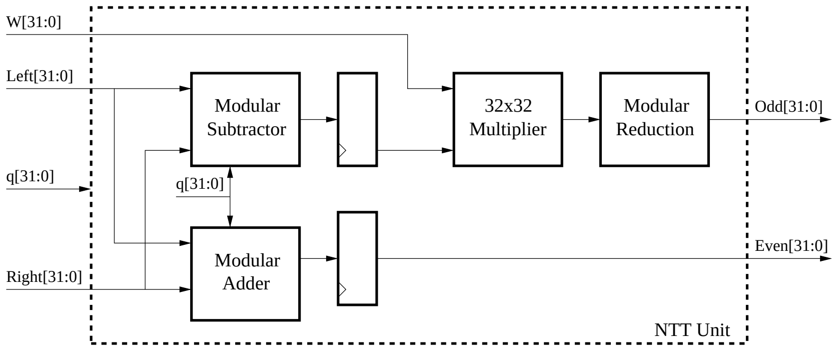

Fig. 4. NTT Unit

Since each polynomial has 1024 coefficients, we decided to utilize 64 of these NTT units and 128 separate BRAMS to hold these coefficients. Each BRAM holds 8 of the coefficients and since we are utilizing an in-place NTT algorithm, after reading a coefficient from a BRAM, we only have $1024/128 = 8$ clock cycles to write back the computed result to its corresponding place. This requirement forced us to design a datapath with at most 6 clock cycle latency. The reason we designed a 5 clock cycle latency datapath is that adding a 6th pipe stage did not improve frequency.

4 Simple Encrypted Arithmetic Library (SEAL)

Simple Encrypted Arithmetic Library (SEAL), which was developed by Cryptography Research Group at the Microsoft Research, is a homomorphic encryption library. It provides an easy-to-use homomorphic encryption library for people in academia and industry.

SEAL uses FV homomorphic encryption scheme for homomorphic operations. Although SEAL uses FV scheme, it performs some operations, such as decryption operation, slightly different than the operations described in Textbook-FV. Secret key generation, public key generation, encryption and decryption operations described in Textbook-FV are already descried in Section 2.1.

In this paper, the proposed work focuses on accelerating the encryption operation in SEAL by implementing the forward/inverse NTT operations used for large degree polynomial multiplications in encryption operation on FPGA. The encryption and decryption operations in SEAL software are explained in detail in the following subsections, 4.1 and 4.2.

4.1 Encryption

Encryption operation in SEAL is implemented the same as the encryption operation in Textbook-FV scheme as shown in Algorithm 7. For the rest of the paper, a variable with a bar over its name represents a polynomial in NTT domain. For example, u and $\overline{u}$ are the same polynomials in polynomial and NTT domains, respectively. In SEAL, public keys, p_0 and p_1, are stored in NTT domain and other polynomials used in encryption operation are stored in polynomial domain. In SEAL, polynomials u, e_1 and e_2 are randomly generated for each encryption operation. Since encryption operation requires polynomial multiplications of u and public keys, p_0 and p_1, the generated u is transformed into NTT domain using NTT operation. After the inner multiplication of $\overline{u}$ and public keys in NTT domain, inverse NTT operation is applied to transform the results from NTT domain to polynomial domain. Finally, necessary polynomial addition operations are performed to generate ciphertexts, c_0 and c_1.

Timing breakdown of the encryption operation in SEAL is shown in Table 1. We performed encryption operation in SEAL 1000 times and obtained the total time for each step for these 1000 operations. The average time for one encryption operation in SEAL is 1.051 ms. As shown in the table, random number generation (RNG) and large degree polynomial multiplication form almost the all of the encryption operation time. The average time for NTT-based large degree polynomial multiplication in one encryption operation is 0.046 ms.

In this paper, NTT-based large degree polynomial multiplications in encryption operation are aimed to be accelerated using FPGA. The design and implementation of efficient RNG for FPGA are beyond the scope of this work and it should be addressed as the future work.

```
Input: $m \in R_t,\ \overline{p_0}, \overline{p_1} \in R_q$
Output: $c_0 = [\Delta \cdot m + p_0 \cdot u + e_1]_q, c_1 = [p_1 \cdot u + e_2]_q$
 1: $u \xleftarrow{\$} R_2$
 2: $p_0u, p_1u =$ NTT_DOUBLE_MULTIPLY $(u, \overline{p_0}, \overline{p_1})$
 3: $e_1 \leftarrow \chi$
 4: $e_2 \leftarrow \chi$
 5: $c_0 = p_0u + e_1 + \Delta \cdot m$
 6: $c_1 = p_1u + e_2$
 7: return $c_0, c_1$
 8:
 9: function NTT_DOUBLE_MULTIPLY$(u, \overline{p_0}, \overline{p_1})$
10:     $\overline{u} =$NTT$(u)$
11:     $\overline{p_0u} = \overline{p_0} \cdot \overline{u}$
12:     $\overline{p_1u} = \overline{p_1} \cdot \overline{u}$
13:     $p_0u =$INTT$(\overline{p_0u})$
14:     $p_1u =$INTT$(\overline{p_1u})$
15:     return $p_0u, p_1u$
16: end function
```

Algorithm 7: Encryption Implementation in SEAL [27]

Table 1. Timing of Encryption Algorithm in SEAL

$\mathbb{Z}_q[x]/(x^{1024}+1)$, q=27-bit, t=8-bit, 128-bit security		
Operation	**Time (μs)**	**Percentage (%)**
$u \leftarrow R_2$	163.6	15.6 %
NTT_DOUBLE_MULTIPLY	46.4	4.4 %
$e_1 \leftarrow \chi$	419.7	39.9 %
$e_2 \leftarrow \chi$	418.4	39.8 %
Others	3.0	0.3 %

4.2 Decryption

Decryption operation described in Textbook-FV requires divide-and-round operation as shown in 4. In order to avoid this costly operation, SEAL uses full RNS variant of Textbook-FV for decryption operation [5].

Decryption operation in SEAL uses ciphertexts, secret key and a redundant modulus $\gamma \in \mathbb{Z}$. In SEAL, secret key, s, is stored in NTT domain and other polynomials used in decryption operation are stored in polynomial domain. Since decryption operation requires polynomial multiplication of c_1 and secret key, s, the ciphertext c_1 is transformed into NTT domain using NTT operation. After the inner multiplication of $\overline{c_1}$ and secret key in NTT domain, inverse NTT operation is applied to transform the result from NTT domain to polynomial domain. Then, c_0 is added into the multiplication result and the resulting polynomial is multiplied with scalar $[\gamma \cdot t]_q$ in polynomial domain. An operation, called *Fast Base Conversion*, is applied to convert polynomial in modulo q to two polynomials in modulo γ and in modulo t. Finaly, the final scaling and multiplication operations are performed to recover decrypted message m.

Timing breakdown of the decryption operation in SEAL is shown in Table 2. We performed decryption operation in SEAL 1000 times and obtained the total time for each step for these 1000 operations. The average time for one decryption operation in SEAL is 0.067 ms.

Table 2. Timing of Decryption Algorithm in SEAL

$\mathbb{Z}_q[x]/(x^{1024}+1)$, q=27-bit, t=8-bit, 128-bit security		
Operation	**Time (μs)**	**Percentage (%)**
NTT_MULTIPLY	29.3	43.5 %
$c_t = (c_1 s + c_0)[\gamma t]_q$	4.4	6.5 %
FASTBCONV	14.8	21.9 %
Others	19.0	28.1 %

```
Input: $c_0, c_1, \overline{s} \in R_q, \gamma \in \mathbb{Z}$
Output: $m \in R_t$
 1: $c_1 s =$ NTT_MULTIPLY $(c_1, \overline{s})$
 2: $c_t = (c_1 \cdot s + c_0)[\gamma \cdot t]_q$
 3: for $m \in \{t, \gamma\}$ do
 4:     $\mathbf{s}^{(m)} \leftarrow$ FASTBCONV$(c_t, q, \{t, \gamma\}) \cdot |-q^{-1}|_m \ mod\ m$
 5: end for
 6: for $i = 0$ to $N$ do
 7:     if $(\mathbf{s}^{(\gamma)}[i] > (\gamma/2))$ then
 8:         $\mathbf{s}^{(\gamma)}[i] = \mathbf{s}^{(\gamma)}[i] - \gamma$
 9:     end if
10:     $m[i] = [\mathbf{s}^{(t)}[i] - \mathbf{s}^{(\gamma)}[i]]_t$
11: end for
12: $m = [m \cdot |-\gamma^{-1}|]_t$
13: return $m$
14:
15: function NTT_MULTIPLY$(c_1 \overline{s})$
16:     $\overline{c_1} =$NTT$(c_1)$
17:     $\overline{c_1 s} = \overline{c_1} \cdot \overline{s}$
18:     $c_1 s =$INTT$(\overline{c_1 s})$
19:     return $c_1 s$
20: end function
21: function FASTBCONV$(x, q, \beta)$
22:     return $(\sum_{i=1}^{k} |x_i . \frac{q_i}{q}|_{q_i} \cdot \frac{q}{q_i}\ mod\ m)_{m \in \beta}$
23: end function
```

Algorithm 8: Decryption Implementation in SEAL [27]

5 The Proposed Design

In this section, we summarize the design techniques we used for our entire framework and briefly explain our optimizations.

5.1 Hardware/Software Co-Design Framework

In order to be able to speed-up homomorphic encryption operations of the SEAL library, we designed a proof-of-concept framework that includes SEAL and an FPGA-based accelerator. To establish communication between the software stack and the FPGA, we utilized RIFFA driver [15], which employs a PCIe connection between the CPU and the FPGA. Resulting framework is shown in Figure 5. Inside SEAL, there is a function:

`ntt_ double_ multiply_ poly_ nttpoly`

which is described as *Perform NTT multiply* $(a * b, a * c)$ *when* b, c *are already in NTT domain.* This function is invoked by the encryption function and it is used to realize NTT_DOUBLE_MULTIPLY operation as explained in Algorithm 7. In our

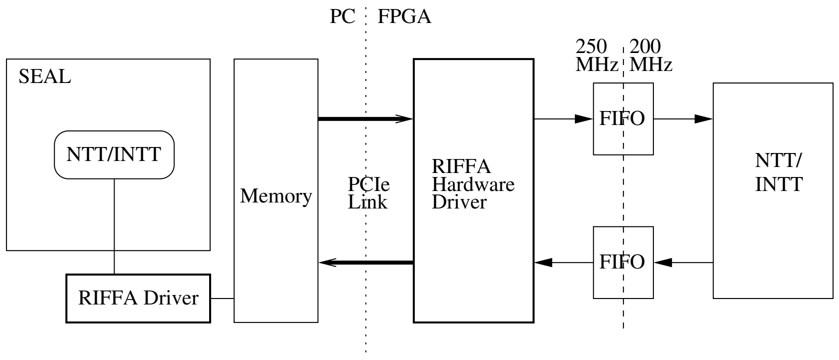

Fig. 5. Hardware/Software Co-Design Framework

modified version of SEAL, this function sends its input data to the connected FPGA and once FPGA returns the computed result, it returns this result to its caller function. One important aspect of this communication is that, since we utilized Direct Memory Access (DMA), necessary data is directly sent from the memory to the FPGA, instead of bringing it to the CPU first. This way, cache of the CPU is not trashed and running this function does not affect the performance of other operations running on the CPU.

For this work, we are using Xilinx Virtex-7 FPGA VC707 Evaluation Kit to realize our framework. This VC707 board includes a PCI Express x8 Gen2 Edge Connector (with a layout for Gen3). XILINX IP Core 7-Series Integrated Block for PCI Express provides a 128-bit interface with a 250 MHz clock, which provides a 4GB/sec bandwidth. As shown in Figure 5, separate FIFO structures are utilized for data input from the RIFFA driver and data output to the RIFFA driver. This approach is utilized to enable a pipelined architecture and maximize performance. For our datapath, we are utilizing a 200 MHz clock to compensate the long critical paths of our design. Although our design is optimized for 1024-degree polynomials, it can easily be modified to realize multiplications for larger degree polynomials.

5.2 Our Hardware

Overall design of our hardware is shown in Figure 6. This hardware employs 64 separate BRAMS for each precomputed parameter (ω, ω^{-1}, Ψ, Ψ^{-1}, Modulus) and 128 separate BRAMS for input U. The multiplier in front of input U is realizing $curr\_\psi = curr\_\psi \cdot \psi$ operation shown in Algorithm 6, as the input is being received from the PCIe link. Therefore, that step of the algorithm does not add any latency to overall NTT operations.

After the hardware computes the NTT of input U, it realizes inner multiplication with P0 and performs inverse NTT on the result. After this operation, the hardware realizes inner multiplication of NTT(U) with P1 and performs

```
Input: Polynomial $U(x) \in \mathbb{Z}_q[x]$ of degree $N-1$, public key ($P0$, $P1$), primitive $2N$-th
    root of unity $\psi \in \mathbb{Z}_q$
Output: Polynomials $U \cdot P0$, $U \cdot P1 \in \mathbb{Z}_q[x]$
 1: for $i$ from 0 by 1 to $N$-1 do
 2:     $U\_\psi[i] = U[i] \cdot \psi^i$
 3: end for
 4: $\overline{U\_\psi}$ =NTT($U\_\psi$)
 5:
 6: for $i$ from 0 by 1 to $N$-1 do
 7:     $\overline{U\_\psi\_P0}[i] = \overline{U\_\psi}[i] \cdot P0[i]$
 8: end for
 9: $U\_\psi\_P0$ =INTT($\overline{U\_\psi\_P0}$)
10:
11: for $i$ from 0 by 1 to $N$-1 do
12:     $\overline{U\_\psi\_P1}[i] = \overline{U\_\psi}[i] \cdot P1[i]$
13: end for
14: $U\_\psi\_P1$ =INTT($\overline{U\_\psi\_P1}$)
15:
16: for $i$ from 0 by 1 to $N$-1 do
17:     $U\_P0[i] = \overline{U\_\psi\_P0}[i] \cdot (\psi^{-i} \cdot N^{-1})$
18: end for
19: for $i$ from 0 by 1 to $N$-1 do
20:     $U\_P1[i] = \overline{U\_\psi\_P1}[i] \cdot (\psi^{-i} \cdot N^{-1})$
21: end for
22: return $U\_P0$, $U\_P1$
```

Algorithm 9: Operation Implemented in Our Design

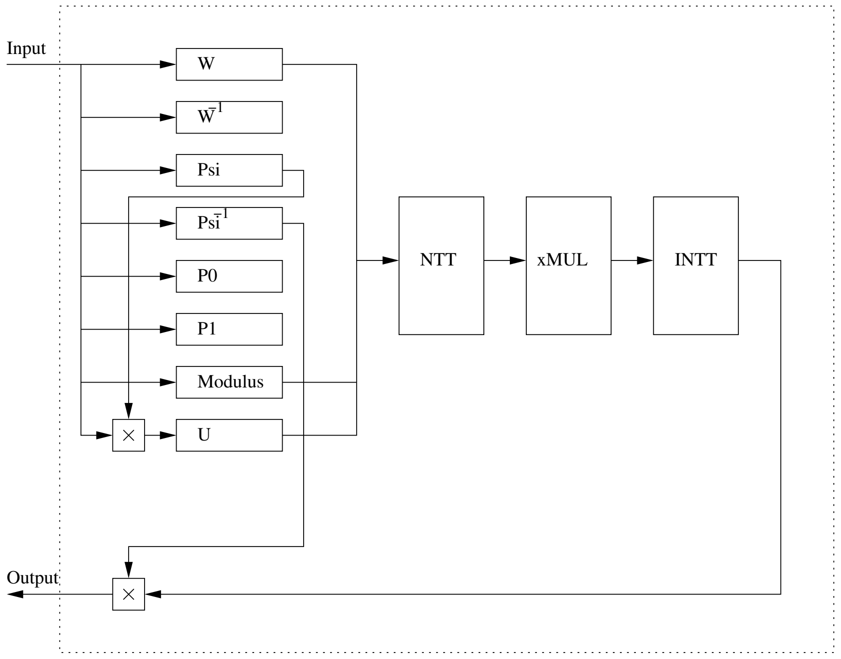

Fig. 6. Our Hardware

Table 3. Comparative table for SPARTAN-6 implementations

Work	**Scheme**	N	q	**LUTs**	**Slice**	**DSP**	**BRAM**	**Period (ns)**	**Latency (μs)**
[6]*	SHE	1024	31-bit	10801	3176	0	0	5.150	40.988
[6]*	SHE	1024	31-bit	6689	2112	4	8	4.154	33.094
[6]*	SHE	1024	31-bit	2464	915	16	14	4.050	32.282
TW**	FV	1024	32-bit	1208	556	14	14	4.727	37.674

* Uses fixed modulus.
** This work

inverse NTT on the result. After inverse NTT operations, necessary multiplications are also realized during the output stage of the overall operation as shown in Algorithm 9.

6 Results and Comparison

In order to be able to present a fair comparison with the state of art in literature, the proposed NTT multiplier is first implemented on a Spartan-6 FPGA (xc6slx100) using Verilog and implementation results are generated using Xilinx

Table 4. Device utilization of the proposed hardware

	Total	**Used**	**Used (%)**
LUTs	303600	33875	11.16%
DFFs	607200	15690	2.58%
RAMB36E1s	1030	227.5	22.09%
DSP48E1s	2800	476	17.00%

Table 5. Comparative table

Work	**Scheme**	**Platform**	N	q **(bits)**	**LUT/ Gate**	**DSP**	**BRAM**	**Freq.**	**Perf. (ms)**
[10]*	GH-FHE	90-nm TSMC	2048	64	26.7 M	–	–	666 MHz	7.750
[21]	LTV	VIRTEX-7	32k	32	219 K	768	193	250 MHz	0.152
[6]*	RLWE	SPARTAN-6	256	21	2829	4	4	247 MHz	0.006
	SHE	SPARTAN-6	1024	31	6689	4	8	241 MHz	0.033
[3]*	HE	SPARTAN-6	1024	17	–	3	2	–	0.100
[22] *	HE	SPARTAN-6	1024	30	1644	1	6.5	200 MHz	0.110
TW**	FV	VIRTEX-7	1024	32	33875	476	227.5	200 MHz	0.00125

* Uses fixed modulus.
** This work

ISE 14.7 with default synthesis option. This small version of our NTT multiplier is designed to be as similar as possible to the one in [6]. The implementation results are shown in Table 3. Our hardware has a clock cycle latency that is almost identical to their design. These results show that our method requires almost half the area with a comparable clock period. Therefore, our method can easily be utilized for any design requiring a generic NTT-friendly prime modulus.

Although SPARTAN-6 family provides fast computations for polynomial multiplication operations, they lack fast I/O infrastructure. Therefore, they are not suitable for accelerator applications requiring high volume of data transfer. From table 3, our hardware achieves $1/37.674\mu s = 26543$ polynomial multiplications per second. A 32-bit coefficient 1024-degree polynomial occupies 32Kbit memory space. Assuming we only have to transfer one polynomial per multiplication to the FPGA, to have a balanced pipelined design, a $829.4 Mb/s$ I/O speed is required, which achieves almost the same result as the CPU implementation. In an accelerator setting, multiple polynomial multipliers need to be instantiated inside the FPGA, which will create a heavy burden on I/O operations.

In our accelerator design, we developed the architecture described in Section 5 into Verilog modules and realized it using Xilinx Vivado 2018.1 tool for the Xilinx Virtex-7 FPGA VC707 Evaluation Kit (XC7VX485T-2FFG1761). The implementation results are summarized in Table 4. There is a plethora of works reported in the literature about multiplication of two large degree polynomials using NTT-based multiplication schemes [10], [21], [6], [3], [22]. Although some of these works also perform different operations such as full encryption and full decryption [10], we only reported the hardware and performance results for large polynomial multiplication part of these works. The works in the literature and the hardware proposed in this paper are reported in Table 5.

Since we target an efficient accelerator design, we implemented our architecture on an FPGA and obtained performance numbers on a real CPU-FPGA heterogeneous application setting. Our XILINX VC707 board has a PCI Express x8 gen2 Edge connector, which can achieve a theoretical 4GB/s connection speed with a 250MHz clock. At this speed, sending a polynomial of degree 1024 with 32-bit coefficients from the CPU to FPGA via DMA takes 1μ s (256 clock cycles). In SEAL software, for encrypt operations, we replaced software-based NTT, Inverse NTT and coefficient multiplication operations with hardware-based operations. In this setting, a pure software implementation yields 46.4μs, and an accelerator-based implementation, including I/O operations, yields 4.3μs latency per polynomial multiplication, which is an almost 11x speedup.

For decryption operation, without pipelining, we achieved $1 + 1.25 + 1 = 3.25\mu s$ for decryption operation, where $1\mu s$ is spent for input, $1.25\mu s$ for polynomail multiplication and another $1\mu s$ is spent for output. Compared to $29.3\mu s$ software performance, we achieved a 9x acceleration. With careful pipelining, overlapping I/O operations with actual polynomial multiplication computations, and assuming one of the operands for the polynomial multiplication operation is already inside the FPGA (a valid assumption for encrypt/decrypt operations for homomorphic applications), we achieved a throughput of almost 800k for degree-1024, 32-bit coefficient polynomial multiplications per second. Compared to $1/29.3\mu s = 34129$ polynomial multiplications per second, we achieved an almost 24x speedup over pure software implementation. Therefore, with pipelining, our hardware can provide almost 3x performance compared to serial implementation.

7 Conclusion

In this paper, we present an optimized FPGA implementation of a novel, fast and highly parallelized NTT-based polynomial multiplier architecture, which is shown to be effective as an accelerator for lattice-based homomorphic cryptographic schemes. To the best of our knowledge, our NTT-based polynomial multiplier has the lowest latency in literature.

For proof of concept, we utilize our architecture in a framework for Fan-Vercauteren (FV) homomorphic encryption scheme, adopting a hardware/software co-design approach, in which NTT operations are offloaded to the accelerator while the rest of operations in the FV scheme are executed in software running on an off-the-shelf desktop computer. We realized the framework on an FPGA connected to the PCIe bus of an off-the-shelf desktop computer and used it to accelerate the FV homomorphic encryption scheme. Our proposed framework operates with SEAL, a homomorphic encryption library developed by Cryptography Research Group at Microsoft Research, and accelerates the encryption and decryption operations in SEAL. We used XILINX VC707 board utilizing a XILINX VIRTEX-7 FPGA for our implementation. We improved the latency of NTT-based large degree polynomial multiplications in encryption operation by almost 11x compared to its pure software implementation. With careful pipelining, overlapping I/O operations with actual polynomial multiplication computations, and assuming one of the operands for the polynomial multiplication operation is already inside the FPGA (valid assumption for encrypt/decrypt operations for homomorphic applications), we achieved a throughput of almost 800k for degree-1024, 32-bit coefficient polynomial multiplications per second.

In this paper, we showed that utilizing a mid-range FPGA as an accelerator for SEAL is very promising for homomorphic encryption operations. Also, since lattice-based cryptography is one of the prominent candidates for post-quantum cryptography applications, our results can be profitably used for hardware implementations of post-quantum cryptographic algorithms when high time performance is required from both latency and throughput perspectives.

As future work, we plan to implement more time-consuming operations used in encryption/decryption operation in SEAL such as RNG on the FPGA board. Alternatively, NTT-based large-degree polynomial multiplications in all operations, not just in encryption, in SEAL can be performed on the FPGA device. We strongly believe that this will lead to better speedup values for the overall computation. Also, RIFFA framework can be modified to serve as a full-duplex communication channel. Although full-duplex communication is not necessary for SEAL library, it can be important for other applications.

References

1. Aguilar-Melchor, C., Barrier, J., Guelton, S., Guinet, A., Killijian, M.O., Lepoint, T.: Nfllib: Ntt-based fast lattice library. In: Sako, K. (ed.) Topics in Cryptology - CT-RSA 2016. pp. 341–356. Springer International Publishing, Cham (2016)
2. Angel, S., Chen, H., Laine, K., Setty, S.: Pir with compressed queries and amortized query processing. Cryptology ePrint Archive, Report 2017/1142 (2017), https://eprint.iacr.org/2017/1142
3. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient fpga implementations of lattice-based cryptography. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). pp. 81–86 (June 2013). https://doi.org/10.1109/HST.2013.6581570
4. Badawi, A.A., Veeravalli, B., Mun, C.F., Aung, K.M.M.: High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. Transactions on Cryptographic Hardware and Embedded Systems **2018**, 70–95 (2018). https://doi.org/10.13154/tches.v2018.i2.70-95
5. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: Avanzi, R., Heys, H. (eds.) Selected Areas in Cryptography – SAC 2016. pp. 423–442. Springer International Publishing, Cham (2017)
6. Chen, D.D., Mentens, N., Vercauteren, F., Roy, S.S., Cheung, R.C.C., Pao, D., Verbauwhede, I.: High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems. IEEE Transactions on Circuits and Systems I: Regular Papers **62**(1), 157–166 (Jan 2015). https://doi.org/10.1109/TCSI.2014.2350431

7. Chen, H., Laine, K., Player, R., Xia, Y.: High-precision arithmetic inhomomorphic encryption. In: Smart, N.P. (ed.) Topics in Cryptology – CT-RSA 2018. pp. 116–136. Springer International Publishing, Cham (2018)
8. de Clercq, R., Roy, S.S., Vercauteren, F., Verbauwhede, I.: Efficient software implementation of ring-lwe encryption. In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 339–344 (March 2015). https://doi.org/10.7873/DATE.2015.0378
9. Dai, W., Sunar, B.: cuhe: A homomorphic encryption accelerator library. In: Pasalic, E., Knudsen, L.R. (eds.) Cryptography and Information Security in the Balkans. pp. 169–186. Springer International Publishing, Cham (2016)
10. Doroz, Y., Ozturk, E., Sunar, B.: Accelerating fully homomorphic encryption in hardware. IEEE Transactions on Computers **64**(6), 1509–1521 (June 2015). https://doi.org/10.1109/TC.2014.2345388
11. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144
12. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford, CA, USA (2009), aAI3382729
13. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing. pp. 169–178. STOC '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1536414.1536440, http://doi.acm.org/10.1145/1536414.1536440
14. Halevi, S., Shoup, V.: Algorithms in helib. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014. pp. 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
15. Jacobsen, M., Freund, Y., Kastner, R.: Riffa: A reusable integration framework for fpga accelerators. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines. pp. 216–219 (April 2012). https://doi.org/10.1109/FCCM.2012.44
16. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security. pp. 124–139. Springer International Publishing, Cham (2016)
17. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing. pp. 1219–1234. STOC '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2213977.2214086, http://doi.acm.org/10.1145/2213977.2214086
18. Lu, W.j., Zhou, J.j., Sakuma, J.: Non-interactive and output expressive private comparison from homomorphic encryption. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 67–74. ASIACCS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3196494.3196503, http://doi.acm.org/10.1145/3196494.3196503
19. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
20. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985)
21. Ozturk, E., Doroz, Y., Savas, E., Sunar, B.: A custom accelerator for homomorphic encryption applications. IEEE Transactions on Computers **66**(1), 3–16 (Jan 2017). https://doi.org/10.1109/TC.2016.2574340

22. Pöppelmann, T., Güneysu, T.: Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In: Hevia, A., Neven, G. (eds.) Progress in Cryptology – LATINCRYPT 2012. pp. 139–158. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
23. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D.: A reconfigurable fabric for accelerating large-scale datacenter services. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. pp. 13–24. ISCA '14, IEEE Press, Piscataway, NJ, USA (2014), http://dl.acm.org/citation.cfm?id=2665671.2665678
24. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing. pp. 84–93. STOC '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1060590.1060603, http://doi.acm.org/10.1145/1060590.1060603
25. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation, Academia Press pp. 169–179 (1978)
26. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact ring-lwe cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014. pp. 371–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
27. Simple Encrypted Arithmetic Library (release 3.1.0). https://github.com/Microsoft/SEAL (Dec 2018), microsoft Research, Redmond, WA.
28. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography – PKC 2010. pp. 420–443. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
29. Wang, J., Arriaga, A., Tang, Q., Ryan, P.Y.A.: Cryptorec: Secure recommendations as a service. CoRR **abs/1802.02432** (2018), http://arxiv.org/abs/1802.02432
30. Yanik, T., Savas, E., Koc, C.K.: Incomplete reduction in modular arithmetic. IEE Proceedings - Computers and Digital Techniques **149**(2), 46–52 (March 2002). https://doi.org/10.1049/ip-cdt:20020235