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

Ahmet Can Mert, Erdinç Öztürk, Erkay Savaş
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
{ahmetcanmert, erdinco, erkays}@sabanciuniv.edu

***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 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 polynomial multiplication operations are offloaded to the accelerator via PCIe bus 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 [1], for the FV encryption scheme, where large degree polynomial multiplications are utilized extensively. The hardware part of the proposed framework targets Xilinx Virtex-7 FPGA device and the proposed framework achieves almost 11x latency speedup for the offloaded operations compared to their pure software implementations. We achieved a throughput of almost 800K polynomial multiplications per second, for polynomials of degree 1024 with 32-bit coefficients.**

***Index Terms*—Number Theoretic Transform, Large-Degree Polynomial Multiplication, Fan-Vercauteren, SEAL, FPGA**

I. 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. The idea of FHE was first introduced in 1978 [2] and it had been an open problem until Gentry constructed the first functioning FHE scheme in 2009 [3]. Since then various practical FHE schemes have been introduced [4], [5], [6]. 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 blocks of the working FHE schemes. Different implementations and constructions were developed to introduce practical hardware and software implementations of FHE schemes, such as [7], HElib [8], NFLlib [9], cuHe [10]. With the motivation of achieving a practical FHE implementation, we focus on improving performance of the most time consuming arithmetic 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 [5].

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) [1], providing a simple and practical software infrastructure using the FV homomorphic encryption scheme for homomorphic applications [5]. SEAL already gained recognition in the literature [11], [12], [13]. 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 FPGA board.

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. 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 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. Besides, 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 input-output (I/O) time. This overhead can be prohibitively high if the nature and 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 between hardware and software components 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 accelerator and SEAL library running on a CPU. 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 Number Theoretic Transform (NTT) operations. For efficient modular arithmetic, we employ lazy reduction techniques as explained in [14]. We also slightly modify the NTT operation loops in order to be able to efficiently parallelize NTT computations. Since 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 cryptographic applications, for security NTT hardware is designed to run in constant time for every possible input combination.
- We introduce a novel modular multiplier architecture for any NTT-friendly prime modulus, which provides comparable time performance to those using special primes.
- 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 $\sim$32 Gbps half-duplex I/O speed. For proof of concept, we accelerate polynomial multiplications utilized in the encryption operations of SEAL. Every time an encryption function is invoked by SEAL, the polynomial multiplications are 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 polynomial multiplication is improved by almost 11x with about a 17% utilization of the VIRTEX-7 resources. With careful pipelining, I/O operations can be overlapped with actual polynomial multiplications on hardware and additional 3x throughput performance can be achieved for pure polynomial multiplication. 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 security assumption.

II. Background

In this section we give definition of the FV scheme as presented in [5] and arithmetic operations utilized in this scheme.

A. FV Homomorphic Encryption Scheme

In [5], the authors present an encryption scheme based on Ring Learning with Errors (RLWE) problem [15]. The RLWE problem is simply a ring based version of the LWE problem [16] and leads to the following encryption scheme as described in [15].

Let the plaintext and ciphertext spaces taken as R_t and R_q, respectively, for some integer $t > 1$. We remark that neither 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 Δ and χ be $\lfloor q/t \rfloor$ and a truncated discrete Gaussian distribution, 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**: $m \in R_t$
$$m = [\lfloor \tfrac{t}{q} [c_0 + c_1 \cdot s]_q \rceil]_t$$

B. 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 modified version of iterative NTT scheme [7] shown in Algorithm 1, which uses the modifications shown in [17]. Inverse NTT (INTT) operation is performed using the same Algorithm 1 with ω^{-1} instead of ω.

C. 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 [14]. In this section we present hardware-friendly constant-time modular arithmetic algorithms. For the rest of the section, we assume a K-bit modulus q. Our modular arithmetic operations compute numbers in the range $[0, 2^K - 1]$, instead of $[0, q - 1]$.

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 q is $2^{K-1} + 1$.

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

Algorithm 1 Modified Iterative NTT

```
Input: Polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$
Input: primitive $n$-th root of unity $\omega \in \mathbb{Z}_q$
Input: $q \equiv 1 \pmod{2n}$, $n = 2^l$
Output: Polynomial $a(x) =$NTT$(a) \in \mathbb{Z}_q[x]$
for $i$ from 1 by 1 to $l$ do
    $m = 2^{l-i}$
    for $j$ from 0 by 1 to $2^{i-1} - 1$ do
        $t = 2 \cdot j \cdot m$
        for $k$ from 0 by 1 to $m - 1$ do
            $curr\_\omega = \omega[2^{i-1}k]$
            $U \leftarrow a[t+k]$
            $V \leftarrow a[t+k+m]$
            $a[t+k] \leftarrow U + V$
            $a[t+k+m] \leftarrow \omega \cdot (U - V)$
        end for
        $\omega \leftarrow \omega \cdot \omega_i$
    end for
end for
return $a$
```

Algorithm 2 Modular Addition Algorithm

```
Input: $A, B, q$ (K–bit positive integers)
Output: $C \equiv A + B \pmod{q}$ (K–bit positive integer)
$T1 = A + B$
$T2 = T1 - q$
$T3 = T1 - 2 \cdot q$
if ($T2 < 0$) then
    $C = T1$
else if ($T3 < 0$) then
    $C = T2$
else
    $C = T3$
end if
```

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

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

Results of Eqn. 1 and Eqn. 2 are $K + 1$-bit numbers, and result of Eqn. 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 in terms of hardware perspective.

2) 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 q is $2^{K-1} + 1$.

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

Algorithm 3 Modular Subtraction Algorithm

```
Input: $A, B, q$ (K-bit positive integers)
Output: $C \equiv A - B \pmod{q}$
$T1 = A - B$
$T2 = T1 + q$
$T3 = T1 + 2 \cdot q$
if ($T2 < 0$) then
    $C = T3$
else if ($T1 < 0$) then
    $C = T2$
else
    $C = T1$
end if
```

Algorithm 4 Word-Level Montgomery Reduction Algorithm

```
Input: $C = A \cdot B$ (a $2K$-bit positive integer)
Input: $q$ (a $K$-bit positive integer)
Input: $\mu = -q^{-1} \pmod{2^w}$ ($w$-bit integer, $w <= K$)
Input: $L = \lceil K/w \rceil$
Output: $Res = C \cdot R^{-1} \pmod{q}$ where $R = 2^{Lw} \pmod{q}$
$T1 = C$
for $i$ from 0 to $L - 1$ do
    $T2 = T1 \pmod{2^w}$
    $T3 = (T2 \cdot \mu) \pmod{2^w}$
    $T1 = \lfloor (T1 + (T3 \cdot q))/2^w \rfloor$
end for
$T4 = T1 - q$
if ($T4 < 0$) then
    $Res = T1$
else
    $Res = T4$
end if
```

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

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

Results of Eqn. 4 and Eqn. 5 are negative numbers, and result of Eqn. 6 is a K-bit positive number. This shows that after a subtraction operation, at most 2 addition operations are required to guarantee 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 in terms of hardware perspective.

3) Modular Multiplication: For our entire hardware, we utilized Montgomery Reduction algorithm [18], for reasons explained in Section IV-A1. 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 in terms of hardware perspective.

III. Simple Encrypted Arithmetic Library (SEAL)

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

Algorithm 5 Encryption Implementation in SEAL [1]

```
Input: $m \in R_t^n,\ \overline{p_0}, \overline{p_1} \in R_q^n$
Output: $c_0 = [p_0 u + e_1 + \Delta \cdot m]_q, c_1 = [p_1 u + e_2]_q$
$u \xleftarrow{\$} R_2$
$p_0 u, p_1 u =$ NTT_DOUBLE_MULTIPLY $(u, \overline{p_0}, \overline{p_1})$
$e_1, e_2 \leftarrow \chi$
$c_0 = [p_0 u + e_1 + \Delta \cdot m]_q$
$c_1 = [p_1 u + e_2]_q$
return $c_0, c_1$
function NTT_DOUBLE_MULTIPLY($u$, $\overline{p_0}$, $\overline{p_1}$)
    $\overline{u}$ =NTT($u$)
    $\overline{p_0 u} = \overline{p_0} \odot \overline{u}$
    $\overline{p_1 u} = \overline{p_1} \odot \overline{u}$
    $p_0 u$ =INTT($\overline{p_0 u}$)
    $p_1 u$ =INTT($\overline{p_1 u}$)
    return $p_0 u, p_1 u$
end function
```

TABLE I
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 (%)
$u \leftarrow R_2$	11.2	7.4 %
NTT_DOUBLE_MULTIPLY	46.4	30.1 %
$e_1, e_2 \leftarrow \chi$	91.1	60.2 %
Others	3.1	2.3 %

library for people in academia and industry. SEAL uses FV homomorphic encryption scheme for homomorphic operations. In this paper, the proposed work focuses on accelerating the encryption operation in SEAL by implementing the polynomial multiplications in encryption operation on FPGA board.

Encryption operation in SEAL is implemented the same as the encryption operation in Textbook-FV scheme as shown in Algorithm 5. 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 I. The average time for one encryption operation in SEAL running on an Intel i9-7900X CPU is 151 μs. The average time for NTT-based large degree polynomial multiplication in one encryption operation is 46.4 μs.

IV. THE PROPOSED DESIGN

In this section, design techniques used for our efficient and scalable NTT-based polynomial multiplier, the design techniques we used for our entire framework and our optimizations are explained. For proof of concept design, we chose to implement a 1024-degree polynomial multiplication architecture targeting a 128-bit security level with 32-bit coefficients. 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 $q \equiv 1 \pmod{2n}$, as shown in Algorithm 4.

A. The Proposed Hardware

1) Modular 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 [14]. Our modular multiplier architecture is optimized for modulus lengths between 22 and 32 bits.

We designed a 32-bit multiplier with 4 DSP blocks and an adder tree. Since we are targeting an 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 18×25-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 32-bit multiplier has pipeline registers between DSP blocks and adder tree. These pipeline registers do not affect the throughput of the overall architecture in terms of clock cycles, improving the overall performance in terms of execution time significantly.

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:

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

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

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

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

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

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

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

Algorithm 6 Word-Level Montgomery Reduction Algorithm modified for NTT-friendly primes

```
Input: $C = A \cdot B$ (a $2K$-bit positive integer, $22 \leq K \leq 32$)
Input: $q$ (a $K$-bit positive integer, $q = q_H \cdot 2^{11} + 1$)
Output: $Res = C \cdot R^{-1} \pmod{M}$ where $R = 2^{33} \pmod{q}$
$T1 = C$
for $i$ from 0 to 2 do
    $T1_H = T1 >> 11$
    $T1_L = T1 \pmod{2^{11}}$
    $T2 = 2's\ complement\ of\ T1_L$
    $carry = T2[10]\ OR\ T1_L[10]$
    $T1 = T1_H + (q_H \cdot T2[10:0]) + carry$
end for
$T4 = T1 - q$
if ($T4 < 0$) then
    $Res = T1$
else
    $Res = T4$
end if
```

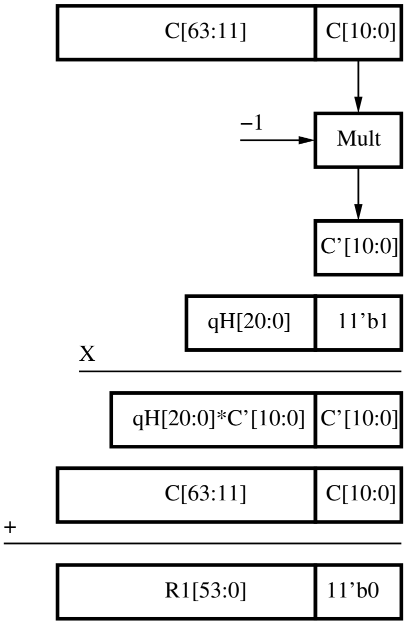

Fig. 1. Flow of operations for Word-Level Montgomery Reduction algorithm modified for NTT-friendly primes

Utilizing this property, we rewrite Montgomery Reduction as shown in Algorithm 6. Flow of operations for Algorithm 6 is shown in Figure 1.

To guarantee that one subtraction at the end of Algorithm 6 is enough, $K < (3 \times 11)$ needs to be satisfied. For $K < (2 \times 11)$, 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 $(4 \times 12) < K < (5 \times 12)$, 5 iterations are required. For $n = 4096$, $w = 13$ and for a modulus of length $(4 \times 13) < K < (5 \times 13)$, 5 iterations are required.

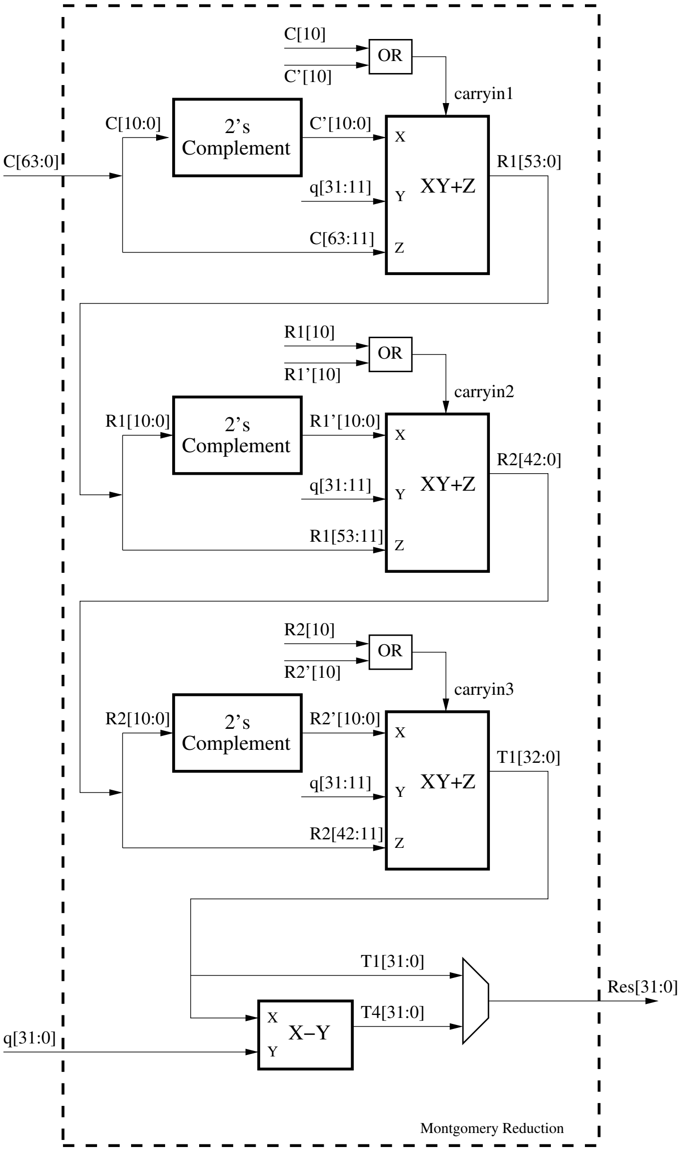

Fig. 2. Word-Level Montgomery Reduction Hardware for NTT-friendly primes

Hardware design for Algorithm 6 is shown in Figure 2. $XY + Z$ 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.

2) NTT unit: To achieve optimized performance for NTT computations, we use the modified version of iterative NTT algorithm shown in Algorithm 1. 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

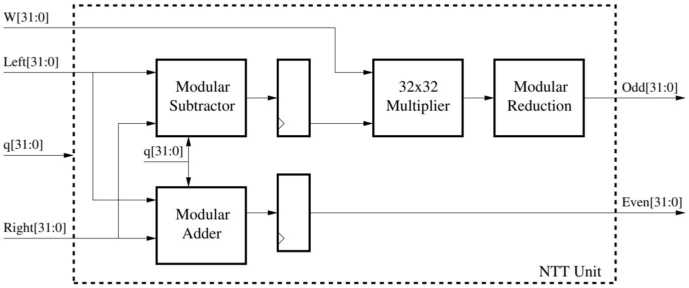

Fig. 3. NTT Unit

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 1, we designed an NTT unit, shown in Figure 3. Latency of this NTT unit is 5 clock cycles.

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 6^{th} pipe stage did not improve frequency.

3) Polynomial Multiplier: Overall design of our hardware is shown in Figure 4. 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 $\hat{u} = u \cdot \Psi$ operation shown in Algorithm 7, 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 p_0 and performs INTT on the result. After this operation, the hardware realizes inner multiplication of $\overline{u}$ with p_1 and performs INTT on the result. After INTT operations, necessary multiplications are also realized during the output stage of the overall operation as shown in Algorithm 7.

B. Hardware/Software Co-Design Framework

In order to be able to speed-up encryption operation 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 [19], which employs a PCIe connection between the CPU and the FPGA. Resulting framework is shown in Figure 5. Inside SEAL, there is `encrypt` function which work as decribed in III. In our 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 operations running on the CPU.

For this work, we are using Xilinx Virtex-7 FPGA VC707 Evaluation Kit which includes a PCIe x8 Gen2 Edge Connector. This provides a 128-bit interface with a 250 MHz clock, which provides a 32 Gbps bandwidth. As shown in

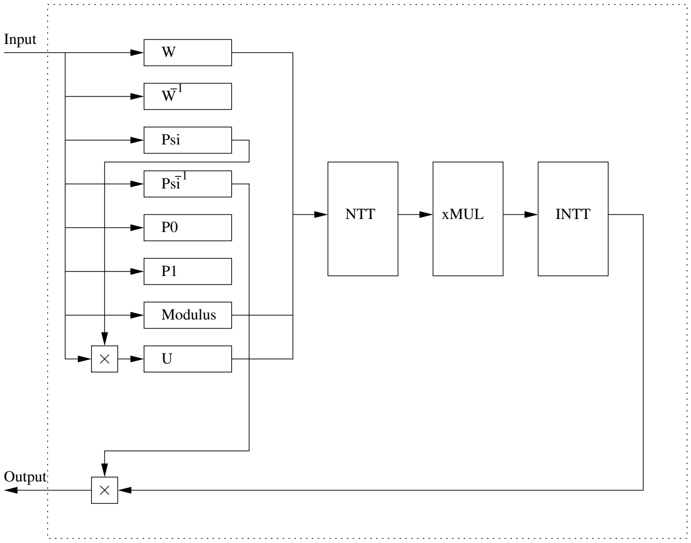

Fig. 4. Our Hardware

Algorithm 7 Operation Implemented in Our Design

```
Input: Polynomial $u \in \mathbb{Z}_q[x]/(x^n+1)$ of degree $n-1$
Input: Public keys ($\overline{p_0}$, $\overline{p_1}$)
Input: Primitive $2n$-th root of unity $\Psi \in \mathbb{Z}_q$
Output: Polynomials $u \cdot p_0$, $u \cdot p_1 \in \mathbb{Z}_q[x]/(x^n+1)$
for $i$ from 0 by 1 to $n-1$ do
    $\hat{u}[i] = u[i] \cdot \Psi^i$
end for
$\overline{u}$ =NTT($\hat{u}$)
for $i$ from 0 by 1 to $n-1$ do
    $\overline{u\_p_0}[i] = \overline{u}[i] \cdot p_0[i]$
    $\overline{u\_p_1}[i] = \overline{u}[i] \cdot p_1[i]$
end for
$u\hat{\_p}_0$ =INTT($\overline{u\_p_0}$)
$u\hat{\_p}_1$ =INTT($\overline{u\_p_1}$)
for $i$ from 0 by 1 to $n-1$ do
    $u\_p_0[i] = u\hat{\_p}_0[i] \cdot (\Psi^{-i} \cdot n^{-1})$
    $u\_p_1[i] = u\hat{\_p}_1[i] \cdot (\Psi^{-i} \cdot n^{-1})$
end for
return $u\_p_0$, $u\_p_1$
```

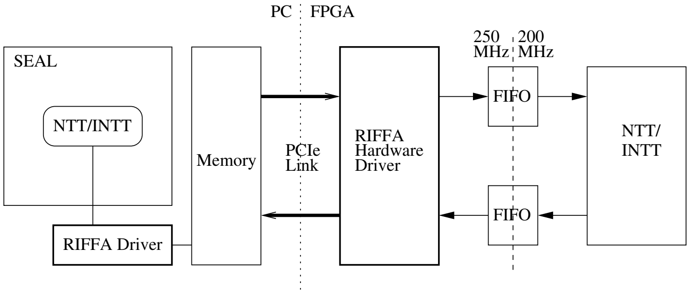

Fig. 5. Hardware/Software Co-Design Framework

TABLE II
COMPARATIVE TABLE

Work	LUTs	Slice	DSP	BRAM	Per.(ns)	Lat.(μs)
[20] *	10801	3176	0	0	5.150	40.988
[20] *	6689	2112	4	8	4.154	33.094
[20] *	2464	915	16	14	4.050	32.282
This	1208	556	14	14	4.727	37.674

* Uses fixed modulus.

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.

V. 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 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 [20]. Both our work and [20] uses polynomial degree of 1024 with 32-bit coefficients. The implementation results are shown in Table II. Per. and Lat. in Table II stand for clock period and latency, respectively. Our hardware has latency that is almost identical to their design and 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 II, our hardware achieves $1/37.674\mu s = 26543$ polynomial multiplications per second. A 32-bit coefficient 1024-degree polynomial occupies 32 Kb memory space. Assuming we only have to transfer one polynomial per multiplication to the FPGA, a 829.4 Mbps 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.

In our accelerator design, we developed the architecture described in Section IV into Verilog modules and realized it using Xilinx Vivado 2018.1 tool for the Xilinx Virtex-7 FPGA VC707 Evaluation Kit (XC7VX485T-2FFG1761). The proposed work uses 33.8K LUTs (11.2%), 15.7K DFFs (2.6%), 227.5 BRAMs (22%) and 476 DSPs (17%). There is a plethora of works reported in the literature about multiplication of two large degree polynomials using NTT-based multiplication schemes [21], [22], [20], [23], [24]. Although some of these works also perform different operations, we only reported the hardware and performance results for polynomial multiplication part of these works. The works in the literature and the work proposed in this paper are reported in Table III.

Although there are other accelerators [25] in the literature performing RLWE encryption and decryption, these works use small parameters and they are not designed for homomorphic operations. Thus, they are not included in the comparison.

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 can achieve a theoretical 32 Gbps I/O 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 $1\mu s$ (256 clock cycles). In SEAL software, for encrypt operations, we replaced polynomial 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 $1 + 1.3 + 2 = 4.3\mu s$ latency per polynomial multiplication, where $1\mu s$, $1.3\mu s$ and $2\mu s$ are spent for input, polynomial multiplication and output, respectively. Compared to software, we achieved a 11x speedup.

Encryption operation in SEAL performs two polynomial multiplications requiring one NTT, two inner multiplication and two INTT operations. Then, we modified our hardware to perform one polynomial multiplication requiring one NTT, one inner multiplication and one INTT. In this case, polynomial multiplication takes $1.25\mu s$. 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 encryption and decryption operations for homomorphic applications), we achieved a throughput of almost 800K for degree-1024, 32-bit coefficient polynomial multiplications per second. Decryption operation in SEAL performs one polynomial multiplication in $29.3\mu s$. Then, the software performance is $1/29.3\mu s = 34129$ polynomial multiplications per second and we achieved an almost 24x speedup over pure software implementation. Therefore, with pipelining, overlapping I/O operations with actual decryption or polynomial multiplication operations, and usng PCIe in full-duplex, our hardware can provide almost 3x performance compared to serial implementation.

TABLE III
COMPARATIVE TABLE

Work	Scheme	Platform	n	q	LUTs/Gate	DSP	BRAM	Freq.	Perf. (ms)
[21] *	GH-FHE	90-nm TSMC	2048	64-bit	26.7 M	–	–	666 MHz	7.750
[22]	LTV	VIRTEX-7	32768	32-bit	219 K	768	193	250 MHz	0.152
[20] *	RLWE	SPARTAN-6	256	21-bit	2829	4	4	247 MHz	0.006
	SHE	SPARTAN-6	1024	31-bit	6689	4	8	241 MHz	0.033
[23] *	HE	SPARTAN-6	1024	17-bit	–	3	2	–	0.100
[24] *	HE	SPARTAN-6	1024	30-bit	1644	1	6.5	200 MHz	0.110
This work	FV	VIRTEX-7	1024	32-bit	33875	476	227.5	200 MHz	0.00125

* Uses fixed modulus.

VI. 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 schemes. To the best of our knowledge, our NTT-based polynomial multiplier has the lowest latency in the literature.

For proof of concept, we utilize our architecture in a framework for FV homomorphic encryption scheme, adopting a hardware/software co-design approach, in which NTT operations are offloaded to the accelerator via PCIe bus 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 operating with SEAL software and the proposed framework accelerates the encryption operation in SEAL. We used Xilinx VC707 board utilizing a Virtex-7 FPGA for our implementation. We improved the latency of NTT-based polynomial multiplications in encryption operation by almost 11x compared to its pure software implementation. We achieved a throughput of almost 800K for degree-1024, 32-bit coefficient polynomial multiplications per second.

REFERENCES

[1] “Simple Encrypted Arithmetic Library (release 3.1.0),” https://github.com/Microsoft/SEAL, Dec. 2018, microsoft Research, Redmond, WA.

[2] R. L. Rivest, L. Adleman, and M. L. Dertouzos, “On data banks and privacy homomorphisms,” *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[3] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford, CA, USA, 2009, aAI3382729.

[4] A. López-Alt, E. Tromer, and V. Vaikuntanathan, “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*, ser. STOC ’12, 2012, pp. 1219–1234.

[5] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” Cryptology ePrint Archive, Report 2012/144, 2012.

[6] N. P. Smart and F. Vercauteren, “Fully homomorphic encryption with relatively small key and ciphertext sizes,” in *Public Key Cryptography – PKC 2010*, P. Q. Nguyen and D. Pointcheval, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 420–443.

[7] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, “Compact ring-lwe cryptoprocessor,” in *Cryptographic Hardware and Embedded Systems – CHES 2014*, L. Batina and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391.

[8] S. Halevi and V. Shoup, “Algorithms in helib,” in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571.

[9] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, “Nfllib: Ntt-based fast lattice library,” in *Topics in Cryptology - CT-RSA 2016*, K. Sako, Ed. Cham: Springer International Publishing, 2016, pp. 341–356.

[10] W. Dai and B. Sunar, “cuhe: A homomorphic encryption accelerator library,” in *Cryptography and Information Security in the Balkans*, E. Pasalic and L. R. Knudsen, Eds. Cham: Springer International Publishing, 2016, pp. 169–186.

[11] W.-j. Lu, J.-j. Zhou, and J. Sakuma, “Non-interactive and output expressive private comparison from homomorphic encryption,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS ’18, 2018, pp. 67–74.

[12] J. Wang, A. Arriaga, Q. Tang, and P. Y. A. Ryan, “Cryptorec: Secure recommendations as a service,” *CoRR*, vol. abs/1802.02432, 2018.

[13] S. Angel, H. Chen, K. Laine, and S. Setty, “Pir with compressed queries and amortized query processing,” Cryptology ePrint Archive, Report 2017/1142, 2017, https://eprint.iacr.org/2017/1142.

[14] T. Yanik, E. Savas, and C. K. Koc, “Incomplete reduction in modular arithmetic,” *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 2, pp. 46–52, March 2002.

[15] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Advances in Cryptology – EUROCRYPT 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[16] 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*, ser. STOC ’05. New York, NY, USA: ACM, 2005, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/1060590.1060603

[17] P. Longa and M. Naehrig, “Speeding up the number theoretic transform for faster ideal lattice-based cryptography,” in *Cryptology and Network Security*, S. Foresti and G. Persiano, Eds. Cham: Springer International Publishing, 2016, pp. 124–139.

[18] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[19] M. Jacobsen, Y. Freund, and R. Kastner, “Riffa: A reusable integration framework for fpga accelerators,” in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 216–219.

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

[21] Y. Doroz, E. Ozturk, and B. Sunar, “Accelerating fully homomorphic encryption in hardware,” *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, June 2015.

[22] E. Ozturk, Y. Doroz, E. Savas, and B. Sunar, “A custom accelerator for homomorphic encryption applications,” *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, Jan 2017.

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

[24] T. Pöppelmann and T. Güneysu, “Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware,” in *Progress in Cryptology – LATINCRYPT 2012*, A. Hevia and G. Neven, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 139–158.

[25] G. Seiler, “Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography,” *IACR Cryptology ePrint Archive*, vol. 2018, p. 39, 2018.