Design and Implementation of Encryption/ Decryption Architectures for BFV Homomorphic Encryption Scheme

Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş, *Member, IEEE*

***Abstract*— Fully homomorphic encryption (FHE) is a technique that allows computations on encrypted data without the need for decryption and it provides privacy in various applications such as privacy-preserving cloud computing. In this article, we present two hardware architectures optimized for accelerating the encryption and decryption operations of the Brakerski/Fan-Vercauteren (BFV) homomorphic encryption scheme with high-performance polynomial multipliers. For proof of concept, we utilize our architectures in a hardware/software codesign accelerator framework, in which encryption and decryption operations are offloaded to an FPGA device, while the rest of operations in the BFV scheme are executed in software running on an off-the-shelf desktop computer. Specifically, our accelerator framework is optimized to accelerate Simple Encrypted Arithmetic Library (SEAL), developed by the Cryptography Research Group at Microsoft Research. The hardware part of the proposed framework targets the XILINX VIRTEX-7 FPGA device, which communicates with its software part via a peripheral component interconnect express (PCIe) connection. For proof of concept, we implemented our designs targeting 1024-degree polynomials with 8-bit and 32-bit coefficients for plaintext and ciphertext, respectively. The proposed framework achieves almost 12× and 7× latency speedups, including I/O operations for the offloaded encryption and decryption operations, respectively, compared to their pure software implementations.**

***Index Terms*— Fan-Vercauteren (FV), FPGA, hardware, number theoretic transform, Simple Encrypted Arithmetic Library (SEAL).**

I. Introduction

FULLY homomorphic encryption (FHE) is the name used for any encryption scheme that allows arithmetic and logical computations directly on ciphertext. This property facilitates privacy-preserving processing of sensitive data, which is a very important and currently unsatisfied demand in cloud computing applications. Since its first introduction in 1978 [1], the idea of FHE has gained widespread attention in the literature and various FHE schemes have been introduced [2]–[4] Although theoretically sound, FHE schemes are not quite ready to be deployed for practical applications due to performance limitations of computer architectures. Applications based on current FHE schemes, which require efficient implementations of computationally expensive mathematical operations, can be orders of magnitude slower than conventional software applications that operate on plaintext data.

For software implementations of FHE, single-core and multicore CPU performances are critical. For single-core performance, frequency of the processor directly affects performance, which cannot be increased substantially with contemporary technology any further. Also, since CPU is required to provide good performance for a diverse set of applications, hardware and/or architectural improvements on a CPU targeting only FHE applications are not feasible. CPU manufacturers increase the performance of a processor with a multicore approach. However, the number of cores that can be included in a multicore architecture is limited due to expensive single-core implementations [5]. While single-core performance of a general-purpose CPU targets sequential algorithms, multicore architectures are more suitable for parallel algorithms.

Most FHE schemes involve a combination of intrinsically serial and highly parallelizable algorithms that will ultimately perform best on heterogeneous architectures [5], which refers to the use of different processing cores to maximize performance. In this article, we propose such a heterogeneous accelerator framework featuring an FPGA core and a CPU to improve the performance of FHE schemes on a system level.

Conventional cryptosystems such as Advanced Encryption Standard (AES) do not have homomorphic property that allows arithmetic computations to be performed directly on ciphertext without decrypting it. On the other hand, homomorphic encryption (HE) schemes, such as Brakerski/Fan-Vercauteren (BFV), allow homomorphic operations directly on the encrypted data and thus enable privacy-preserving processing of information, especially in the context of cloud computing whereby privacy is a pressing concern. Moreover, HE schemes are patently slow; hence, the case for acceleration is much stronger than conventional cryptosystems. With an ever-increasing demand for privacy in cloud computing applications, acceleration of homomorphic encryption schemes is already an important research area.

There is still ongoing research and race to improve the performance of arithmetic building blocks of the working FHE schemes. Different implementations and architectures were

Manuscript received May 2, 2019; revised August 9, 2019; accepted September 3, 2019. Date of publication October 11, 2019; date of current version January 21, 2020. *(Corresponding author: Ahmet Can Mert.)*

The authors are with the Faculty of Engineering and Natural Sciences, Sabanci University, 34956 Istanbul, Turkey (e-mail: ahmetcanmert@sabanciuniv.edu; erdinco@sabanciuniv.edu; erkays@sabanciuniv.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2943127

developed to facilitate practical FHE schemes: hardware architectures in [6] and [7], software libraries such as HElib [8] and NFLlib [9], and GPU accelerators such as cuHe [10]. With a similar motivation, we aim to obtain a framework to accelerate the FV encryption scheme for homomorphic operations [4]. We focus on improving FPGA performance of the most time-consuming arithmetic building block of many FHE schemes in the literature: large-degree polynomial multiplication. The framework running on our heterogeneous architecture offloads not only polynomial multiplications but also entire encryption and decryption operations onto the FPGA core in order to minimize the communication cost between FPGA core and the CPU.

Cryptography Research Group at Microsoft Research developed Simple Encrypted Arithmetic Library (SEAL) [11], providing a simple and practical software infrastructure using the BFV homomorphic encryption scheme for homomorphic applications [4]. SEAL already gained recognition in the literature [12]–[14]. Wang *et al.* [12] propose a privacy-preserving recommendation service utilizing the SEAL library for homomorphic operations. The GPU implementation in [13] 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 remote server without revealing which element is downloaded [14].

Our accelerator framework, while offloading highly parallelizable encryption and decryption operations entirely on the FPGA core, leaves the rest of operations of SEAL intact in software. By deploying our framework, any cloud architecture utilizing SEAL for FHE applications can improve its performance by utilizing an FPGA device next to the CPU, without having to implement the entire FHE library in the FPGA.

Our contribution in this article is listed as follows.

1) We analyze the iterative [15] and the four-step Cooley-Tukey [16] algorithms for number theoretic transform (NTT) operation and design two novel, highly parallelized hardware architectures based on these algorithms.
2) We evaluate the results of our FPGA implementations of the two hardware architectures in terms of time and area and compare them against similar works in the literature.
3) We demonstrate that our hardware architectures using the novel modular multiplier algorithm for any NTT-friendly prime modulus, introduced in our preliminary work [17], provide comparable time performance to those using special primes in the literature.
4) We propose an accelerator framework, including a high-performance FPGA device, connected to a host CPU. The framework interfaces the CPU and the FPGA via a fast peripheral component interconnect express (PCIe) connection, achieving a ∼32-Gb/s half-duplex I/O speed. The framework is used to accelerate encryption and decryption operations of SEAL. Every time an `encrypt` or `decrypt` function is invoked by SEAL, the computation is offloaded to the FPGA device via the PCIe connection. Our design utilizes a construction targeting 128-bit security level. Including the time spent on I/O, the latencies of the offloaded encryption and decryption operations are improved by 12× and 7×, respectively, compared to their pure software implementations on SEAL running on an Intel i9-7900X CPU.
5) As the 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 and lattice-based cryptosystem.
6) Even though the proposed framework is used to report the accelerations of only the encryption and decryption operations of the BFV homomorphic encryption scheme in our submission, it can be profitably employed to accelerate other operations in homomorphic applications. Indeed, one of the most important contributions of our work is to propose a high-performance polynomial multiplier that can accelerate the multiplication of two very large-degree polynomials, which constitutes the computational bottleneck of all homomorphic operations in the BFV scheme or other schemes. We expect even higher speedup values when our framework is used for homomorphic applications where the communication between CPU and FPGA is expectedly less. This last point is expressed in the conclusion of our submission. Also, in comparative table, we included the time and space performance of the proposed polynomial multiplier with higher ring degrees and larger moduli, which can be used for homomorphic applications.

II. Background

In this section, we give the definition of the BFV scheme as presented in [4] and arithmetic operations utilized in our implementations.

A. BFV Homomorphic Encryption Scheme

Fan and Vercauteren [4] extend Brakerski's encryption scheme [18] from learning with errors (LWE) setting to ring LWE (RLWE) setting [19]. The RLWE problem is simply a ring-based version of the LWE problem [20], which leads to the following encryption scheme as described in [19].

Let the plaintext and ciphertext spaces be the rings R_t and R_q, respectively, for some integers $q > t > 1$. We remark that neither q nor t have to be primes nor that t and q have to be coprime. Let $\lfloor . \rfloor$, $\lfloor . \rceil$ and $[.]_q$ represent flooring, round to nearest integer, and the reduction by modulo q operations, respectively. Let also $\mathbf{a} \xleftarrow{\$} \mathbf{S}$ indicate that $\mathbf{a}$ is uniformly sampled from the set $\mathbf{S}$, χ be a truncated discrete Gaussian distribution, and $\Delta = \lfloor q/t \rfloor$. For the rest of this article, n, q, and t denote the degree of polynomial modulus, the prime used as ciphertext modulus, and the integer used as plaintext modulus, respectively. Secret and public key generation and encryption and decryption operations described in textbook-BFV are shown in the following.

1) *SecretKeyGen*: $s \xleftarrow{\$} R_2^n$.
2) *PublicKeyGen*: $a \xleftarrow{\$} R_q^n$ and $e \leftarrow \chi$.

$$(p_0, p_1) = ([-(a \cdot s + e)]_q, a).$$

Algorithm 1 Modified Iterative NTT Algorithm

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

Algorithm 2 Four-Step Cooley-Tukey NTT Algorithm

Input: $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$
Input: primitive n-th root of unity $\omega \in \mathbb{Z}_q$, $n = n_1 \cdot n_2$
Output: $\overline{a}(x) = \mathtt{NTT}(a) \in \mathbb{Z}_q[x]/(x^n+1)$
$b = [a_0, a_1, \ \ldots\ , a_{n_2-1}]$
for i from 0 by 1 to n_2-1 **do**
  $b[i] \leftarrow \mathtt{NTT}_{n_1}(a[i])$
end for
for i from 0 by 1 to n_1-1 **do**
  for j from 0 to n_2-1 **do**
    $b[i][j] \leftarrow b[i][j] \cdot \omega^{ij}$
  end for
end for
$b \leftarrow$transpose(b)
for i from 0 by 1 to n_1-1 **do**
  $a[i] \leftarrow \mathtt{NTT}_{n_2}(b[i])$
end for
return $a \leftarrow [a_0, a_1, \ \ldots\ , a_{n_1-1}]$

3) *Encryption*: $m \in R_t$, $u \xleftarrow{\$} R_2^n$ and $e_1, e_2 \leftarrow \chi$.

$$(c_0, c_1) = ([\Delta \cdot m + p_0 \cdot u + e_1]_q, [p_1 \cdot u + e_2]_q).$$

4) *Decryption*: $m = [\lfloor (t/q)[c_0 + c_1 \cdot s]_q \rceil]_t$.

B. Number Theoretic Transform

NTT is a discrete Fourier transform defined over ring $R_q = \mathbb{Z}_q/\phi_m(x)$, where $\phi_m(x)$ is the mth cyclotomic polynomial. The forward NTT operation takes an $n-1$ degree polynomial, $A(x) = \sum_{i=0}^{n-1} a_i x^i$, in R_q and produces another polynomial $\mathcal{A}(x) = \sum_{i=0}^{n-1} \mathcal{A}_i x^i$ of degree $n-1$. The coefficients $\mathcal{A}_i$ are defined as $\mathcal{A}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ over $\mathbb{Z}_q$, where $\omega \in \mathbb{Z}_q$ is called as the twiddle factor. The twiddle factor is a primitive nth root of unity in $\mathbb{Z}_q$, satisfying the conditions $\omega^n \equiv 1 \pmod q$ and $\forall i < n$, $\omega^i \not\equiv 1 \pmod q$, where $q \equiv 1 \pmod n$. Similarly, the inverse NTT (INTT) operation can be computed as $a_i = n^{-1} \sum_{j=0}^{n-1} \mathcal{A}_j \omega^{-ij}$ in $\mathbb{Z}_q$.

In this article, we utilize two different NTT schemes: the modified version of the iterative NTT scheme shown in Algorithm 1 [15] and the four-step Cooley-Tukey NTT scheme shown in Algorithm 2 [16].

C. Polynomial Multiplication

The fundamental arithmetic operation in the BFV scheme, during the encryption and decryption operations, is the multiplication of two polynomials of very large degrees. More specifically, we need to multiply two polynomials $A(x)$ and $B(x)$, namely, $\sum_{i=0}^{n-1} a_i x^i$ and $\sum_{i=0}^{n-1} b_i x^i$, over the ring of polynomials $\mathbb{Z}_q[x]/\phi(x)$, where q is an odd integer and the degree of the reduction polynomial, $\phi(x)$, is n. The classical techniques for polynomial multiplication such as the schoolbook polynomial multiplication have quadratic complexity and they are not efficient for large values of n. Instead, NTT-based polynomial multiplication achieves a quasi-linear complexity.

The multiplication of two polynomials, $A(x)$ and $B(x)$, of degree $n-1$ can be calculated using NTT. First, NTT is applied to polynomials $A(x)$ and $B(x)$, then their coefficient-wise multiplication is performed in $\mathbb{Z}_q$, and INTT is applied to obtain the resulting polynomial $C(x)$ as shown in the following equation, where $\odot$ represents coefficient-wise multiplication in $\mathbb{Z}_q$:

$$C(x) = \mathrm{INTT}_{2n}(\mathrm{NTT}_{2n}(A(x)) \odot \mathrm{NTT}_{2n}(B(x))). \quad (1)$$

Since the polynomials $A(x)$ and $B(x)$ are degrees of $n-1$, the resulting polynomial $C(x)$ should have degrees of $2n-1$. Therefore, the polynomials $A(x)$ and $B(x)$ should be padded with n zeros prior to NTT in order to have $2n$ coefficients. Finally, the resulting polynomial $C(x)$ is reduced by applying reduction modulo $\phi(x)$ to it.

When the reduction polynomial $\phi(x)$ has the form of x^n+1, NTT is called as Fermat theoretic transform and a special technique called negative wrapped convolution can be exploited. It avoids doubling of input polynomials with zero-padding and reduction modulo x^n+1 after the polynomial multiplication at the cost of extra $3n$ multiplications. In this case, the coefficients of input polynomials, $A(x)$ and $B(x)$, are multiplied with the powers of $\Psi \in \mathbb{Z}_q$, which is a primitive $2n$th root of unity, where $q \equiv 1 \pmod{2n}$ and $\Psi^2 \equiv \omega \pmod q$, prior to NTT. After the NTT, the coefficient-wise multiplication is performed in $\mathbb{Z}_q$, then INTT is applied, and finally, the coefficients of resulting polynomial are multiplied with the powers of Ψ^{-1} in $\mathbb{Z}_q$ to obtain the resulting polynomial $C(x)$.

III. Simple Encrypted Arithmetic Library

SEAL provides an easy-to-use homomorphic encryption library and provides the academia as well as industry with the practical use of the homomorphic operations. SEAL implements two different encryption schemes: BFV and the Cheon-Kim-Kim-Song (CKKS). Since our hardware architectures realize the encryption and decryption operations of the BFV scheme in SEAL, the CKKS scheme will not be mentioned in this section.

Algorithm 3 Encryption Implementation in SEAL [11]

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

Encryption operation of the BFV in the SEAL is implemented the same way as the encryption operation in textbook-BFV as shown in Algorithm 3. Henceforth, we drop the polynomial notation for the elements of R_q and use small case variable names that are said to be in the polynomial domain. Also, a variable with a bar over it represents a ring element after NTT is applied and says it is in NTT domain. For example, u and $\overline{u}$ denote the same ring element in polynomial and NTT domains, respectively. In SEAL, public keys, $\overline{p_0}$ and $\overline{p_1}$, are stored in NTT domain and other ring elements used in the encryption, u, e_1, e_2, and m, are stored in polynomial domain. The ciphertext pair, c_0 and c_1, is also stored in the polynomial domain after encryption operation. In SEAL, ring elements u, e_1 and e_2, are randomly generated for each encryption operation, and the SEAL uses hardware-based AES in counter mode for pseudo-randomness by default. SEAL employs encoding schemes to convert plaintexts from its integer representation to polynomial representation, which is needed for the encryption operation. Therefore, the plaintext input m in Algorithm 3 is encoded as an element of R_t and stored in the polynomial domain.

To improve its performance, the decryption operation of the BFV scheme in SEAL, shown in Algorithm 4, is implemented slightly different from the textbook-BFV, which requires division and rounding operations. In order to avoid these costly operations, SEAL uses a full residue number system (RNS) variant of textbook-BFV for decryption operation [21], which requires base conversion as shown in Step 4 of Algorithm 4. This optimization is also used in our hardware realization. Decryption operation in SEAL uses ciphertexts, secret key, and a redundant modulus $\gamma \in \mathbb{Z}$. In SEAL, secret key, $\overline{s}$, is stored in the NTT domain.

Timing breakdowns of the encryption and decryption implementations in SEAL are given in Table I. The average time for one encryption and decryption operation in SEAL running on an Intel i9-7900X CPU is 151 and 65.7 μs, respectively.

Algorithm 4 Decryption Implementation in SEAL [11]

Input: $c_0, c_1, \overline{s} \in R_q^n, \gamma \in \mathbb{Z}, \gamma > q,\ \gcd(\gamma, q) = 1$
Output: $m \in R_t^n$
1: $c_1 s =$ NTT_MULTIPLY $(c_1, \overline{s})$
2: $c_t = (c_1 s + c_0) \cdot [\gamma \cdot t]_q$
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 from 0 by 1 to $n-1$ **do**
7: **if** $(\mathbf{s}^{(\gamma)}[i] > (\gamma/2))$ **then**
8: $m[i] = [\mathbf{s}^{(\gamma)}[i] - \mathbf{s}^{(\gamma)}[i] + \gamma]_t$
9: **else**
10: $m[i] = [\mathbf{s}^{(\gamma)}[i] - \mathbf{s}^{(\gamma)}[i]]_t$
11: **end if**
12: **end for**
13: **return** $m \leftarrow [m \cdot [\gamma^{-1}]_t]_t$
14: **function** NTT_MULTIPLY($c_1, \overline{s}$)
15: $\overline{c_1} = \mathtt{NTT}(c_1)$
16: $c_1 s = \mathtt{INTT}(\overline{c_1} \odot \overline{s})$
17: **return** $c_1 s$
18: **end function**
19: **function** FASTBCONV(x, q, β)
20: **return** $(\sum_{i=1}^{k} [x_i . \frac{q_i}{q}]_{q_i} \cdot \frac{q}{q_i} \pmod m)_{m \in \beta}$
21: **end function**

TABLE I
TIMING OF ENCRYPTION AND DECRYPTION IMPLEMENTATIONS IN SEAL

$\mathbb{Z}_q[x]/(x^{1024}+1)$, q=27-bit, t=8-bit, 128-bit security		
Operation	**Time (μs)**	**Percentage (%)**
Encryption		
$u \leftarrow R_2$	11.2	7.4 %
NTT_DOUBLE_MULTIPLY	45.6	30.1 %
$e_1, e_2 \leftarrow \chi$	91.1	60.2 %
Others	3.1	2.3 %
Decryption		
NTT_MULTIPLY	28.8	43.2 %
FASTBCONV	19.5	29.2 %
Others	17.4	27.6 %

IV. PROPOSED ACCELERATOR FRAMEWORK

In this section, we explain the two proposed architectures, summarize the design techniques used in our accelerator framework, and briefly explain our optimizations. The proposed architectures target 128-bit security level, using degree-1024 polynomials with 8-bit and 32-bit coefficients for plaintext and ciphertext, respectively.

A. Proposed Encryption/Decryption Hardware

Here, we first present our Montgomery modular multiplier hardware architecture and its implementation. We then explain two encryption/decryption hardware architectures implementing the iterative and the four-step Cooley-Tukey NTT algorithms for polynomial multiplication operation, respectively. Henceforth, they are shortly referred to as the iterative hardware and the four-step hardware, respectively.

Algorithm 5 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: $Res = C \cdot R^{-1} \pmod{q}$ where $R = 2^{33} \pmod{q}$

1: $T1 = C$
2: **for** i from 0 to 2 **do**
3: $\quad T1_H = T1 >> 11$
4: $\quad T1_L = T1 \pmod{2^{11}}$
5: $\quad T2 = 2's\ complement\ of\ T1_L$
6: $\quad carry = T2[10] \ \vee \ T1_L[10]$
7: $\quad T1 = T1_H + (q_H \cdot T2[10:0]) + carry$
8: **end for**
9: $T4 = T1 - q$
10: **if** $(T4 < 0)$ **then** $Res = T1$ **else** $Res = T4$

1) Montgomery Modular Multiplier: A fast and efficient modular multiplier plays an extremely important role in the optimization of large degree polynomial multiplications as multiplication with modulo q is the computational bottleneck for the BFV scheme. Our modular multiplier unit utilizes the word-level version of the Montgomery reduction algorithm [22] [17], which runs in constant time, with a lazy reduction approach explained in [23].

The architecture is optimized for modulus lengths between 22 and 32 bits. The design has a fully pipelined, 32-bit integer multiplier unit consisting of mainly 4 DSP blocks and an adder tree with 2 clock cycles (cc) latency. Each input of the multiplier is divided into two 16-bit words and these words are multiplied with each other using 4 DSP blocks in one cc, and the 64-bit result is calculated using the adder tree. The pipeline registers in the 32-bit multiplier do not affect the throughput of the overall architecture in terms of cc. In fact, it plays a key role in improving the overall performance of NTT operation in terms of execution time significantly.

After a 32-bit multiplication operation, the 64-bit result needs to be reduced modulo q. For a scalable architecture, we modify the word-level version of the Montgomery reduction algorithm to achieve a fast and efficient modular reduction operation. For efficiency, we utilize the property of all NTT-friendly primes: $q \equiv 1 \pmod{2n}$. Any NTT-friendly prime q be written as $q = q_H \cdot 2^{\log_2 2n} + 1$. Since we select $n = 1024$ for our implementation, we have $q = q_H \cdot 2^{11} + 1$. If we select word size $\omega = 11$ for the Montgomery modular reduction algorithm, we have $\mu = -q^{-1} \pmod{2^{11}} \equiv -1 \pmod{2^{11}}$. Utilizing this property, we can rewrite Montgomery reduction as shown in Algorithm 5. To guarantee that one subtraction at the end of Algorithm 5 suffices, the condition $K < (3 \cdot 11)$ needs to be satisfied. On the other hand, for $K < (2 \cdot 11)$, two iterations are required instead of three. Our algorithm can easily be modified to scale for larger n values. For example, for $n = 2048$, $\omega = 12$, and for a modulus of length $(4 \cdot 12) < K < (5 \cdot 12)$, five iterations are required.

Our hardware architecture for reduction operation in Algorithm 5 is shown in Fig. 1. $XY + Z$ is a multiply-accumulate operation, which can be realized using the FPGA DSP blocks. The proposed hardware with 3 DSP blocks has 3 cc latency.

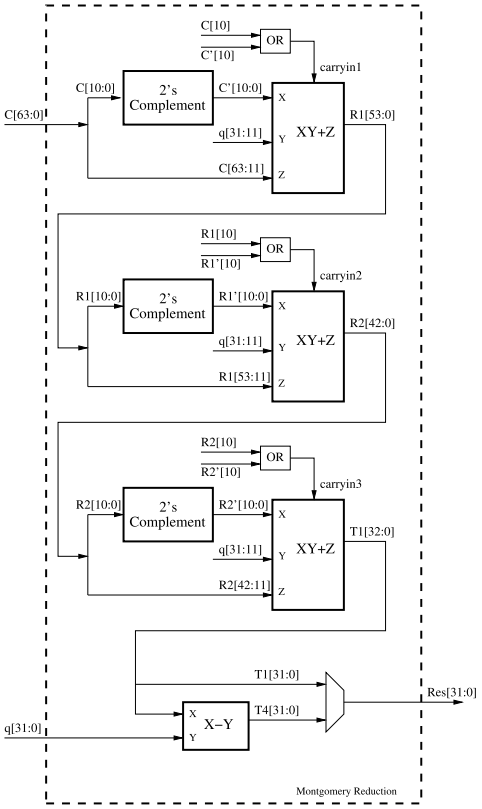

Fig. 1. Word-level Montgomery reduction algorithm modified for NTT-friendly primes.

Algorithm 5 takes $A \cdot B$ as input and calculates $A \cdot B \cdot R^{-1} \pmod{q}$. In order to eliminate factor R^{-1} from the multiplication result, either the output or one of the inputs should be multiplied with R in $\mathbb{Z}_q$. In the proposed hardware, however, instead of inputs of the encryption and decryption operation, other precomputed constants such as ω and Ψ^{-1} used as multiplicands during the operations are multiplied with R or the powers of R prior to being loaded to the FPGA.

2) Four-Step Hardware: The four-step hardware architecture implements Algorithm 2 for polynomial multiplication operation. Encryption and decryption implementations in SEAL use three different arithmetic operations, namely, NTT-based polynomial multiplication and coefficient-wise modular multiplication and addition. Therefore, we designed three arithmetic units: an NTT unit (NU) for predefined ring degree ($n_1 = n_2 = 32$), coefficient-wise modular multiplication, and addition units in our architecture.

Modular multiplication and NTT operations require similar hardware logic, and a reconfigurable hardware could be designed for performing both operations. However, this would require extra control logic routing throughout the device and reduce the performance. In addition, consecutive modular multiplication and NTT operations would not be pipelined efficiently. Therefore, for performance reasons, we use two separate units for modular multiplication and NTT operations.

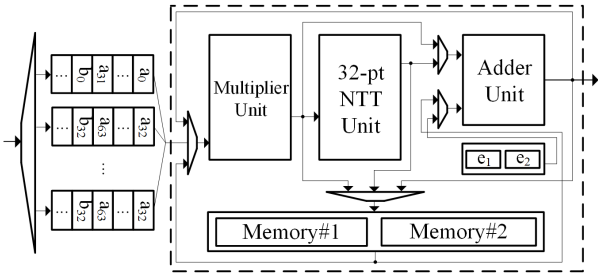

Fig. 2. Four-step hardware for encryption and decryption.

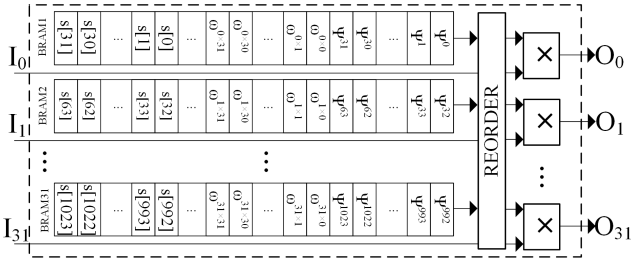

Fig. 3. Multiplier unit of four-step hardware.

In order to make the NTT hardware efficient and reusable, n_1 and n_2 shown in Algorithm 2 are chosen as 32 for $n = n_1 n_2 = 1024$. Therefore, we can use the same 32-point NU for both n_1-point and n_2-point NTT operations.

a) 32-point NTT unit: The 32-point NU uses Cooley-Tukey NTT algorithm [16] for implementing NTT as proposed in [24]. The algorithm takes the input, splits it into two halves, and performs the half-sized NTT operation on the halves, and finally, performs a reconstruction operation to combine the result of the two half-sized NTT operations into the result of the full-sized NTT operation. The reconstruction operation consists of a set of additions and subtractions in series with a set of multiplications. This is known as the divide-and-conquer approach that can be applied recursively to smaller parts.

The 32-point NU has 16 2-point NUs and 4 reconstruction stages. A 2-point NU takes A and B as inputs and calculates $A+B \pmod q$ and $A-B \pmod q$. The NU is pipelined and its latency is 28 clock cycles. The four reconstruction stages have 8, 12, 14, and 15 modular multipliers, respectively.

b) Overall design: The overall design of the four-step hardware architecture is shown in Fig. 2, which basically consists of a 32-point NU, a 32-point coefficient-wise modular multiplier, and modular addition units. In addition to the arithmetic units, 32 separate block random access memories (BRAMs) are used for storing precomputed, e_1 and e_2 (see Algorithm 3). The hardware also uses two memory blocks, each consisting of 32 BRAMs, for storing intermediate values during computations. Each memory block can perform transpose operation as proposed in [25] besides read/write operations. Since the four-step NTT algorithm treats its input as matrix and applies NTT on the columns of the matrix, the four-step hardware uses 32 input FIFOs for retrieving the inputs in the correct order. Thus, the four-step hardware can take 32 input coefficients per clock as in Algorithm 2. A similar structure utilizing 32 output FIFOs is also used at the output of the hardware. However, it is not shown for simplicity.

The four-step hardware employs 32 separate BRAMs within the multiplier unit for storing precomputed parameters, $\overline{p_0}$, $\overline{p_1}$, and $\overline{s}$, and the powers of ω, Ψ, and Ψ^{-1}, as shown in Fig. 3. Each a precomputed parameter has $n = 1024$ elements. The first 32 elements of each parameter are stored in the first BRAM. Similarly, the second 32 elements of each parameter are stored in the second BRAM, and so on, as shown in Fig. 3. These parameters are stored in the same order as the order the four-step hardware takes its inputs, which makes address generation easier. The outputs of 32 BRAMs are connected to the inputs of 32 modular multipliers in the multiplier unit. When a polynomial needs to be multiplied with the one of the precomputed values, necessary addresses are generated to read the precomputed data from 32 BRAMs to the inputs of modular multipliers. Since INTT is also performed in the same unit with a different input order as explained in [10], there is a reordering unit in the multiplier unit.

The polynomial multiplications with public and secret keys are performed in a slightly different way from its description in II-C. Since the public key in the encryption operation and the secret key in the decryption operation are already in the NTT domain, none of them requires NTT. Therefore, the proposed hardware assumes that one of the operands in polynomial multiplication is already in the NTT domain, which is a valid assumption for encrypt/decrypt operations for homomorphic applications, and it performs only one NTT and one INTT for polynomial multiplications.

The hardware starts the encryption operation by multiplying input u with the powers of Ψ in the multiplier unit, which takes $32+6 = 38$ cc. The multiplier unit takes 32 coefficients as inputs per cycle and produces 32 outputs per cycle with 6 cc latency. The resulting polynomial is sent to the NU. In parallel to the $\mathtt{NTT}(u)$ operation, $m\Delta + e_1$ is computed using the multiplier and the adder units. Then, the result of $m\Delta + e_1$ is stored in the first memory block. It should be noted that since the proposed hardware is pipelined, the results of the multiplier unit are directly sent to the NU as soon as the first 32 outputs are calculated. The pipeline overlaps consecutive coefficient-wise multiplication and NTT operations and reduces the overall latency.

The NU performs 32 32-point NTT operations in $28+32 = 60$ cc and the resulting coefficients are stored in the second memory block for the subsequent transpose operation. After the results of the last 32-point NTT are written into the memory block, 32 coefficients are read per cycle from the memory block and sent to the multiplier unit for multiplication with the twiddle factors. The multiplier unit performs multiplication operations and the resulting coefficients are directly sent to the NU, which completes in 60 cc. In total, the proposed hardware finishes $\mathtt{NTT}(u)$ in 140 cc.

Then, $\overline{u}$ is sent to the multiplier unit that performs the multiplications of $\overline{u}$ with $\overline{p_0}$ and $\overline{p_1}$ in $64 + 6 = 70$ cc. The resulting polynomials, $\overline{p_0 u}$ and $\overline{p_1 u}$, are sent to the NU for INTT. Since INTT requires different input ordering, polynomials, $\overline{p_0 u}$ and $\overline{p_1 u}$, are stored in the second memory block after the multiplication for the input reordering. INTT

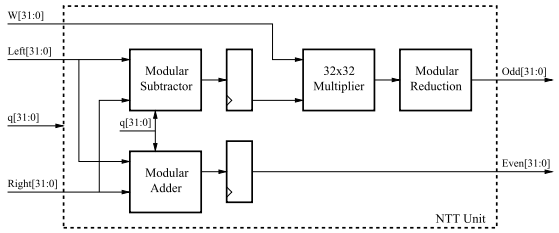

Fig. 4. Iterative NU.

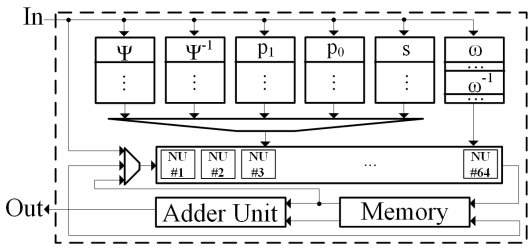

Fig. 5. Iterative hardware for encryption and decryption.

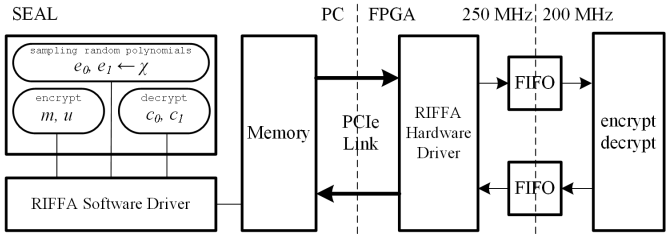

Fig. 6. Hardware/software codesign framework.

of $\overline{p_0 u}$ and $\overline{p_1 u}$ is performed in $140 + 32 = 172$ cc, and the resulting polynomials, $p_0 u$ and $p_1 u$, are directly sent to the multiplier unit for multiplication with the powers of Ψ^{-1}. Finally, $p_0 u$ and $p_1 u$ are directly sent to the adder unit for addition with $\Delta m + e_1$ and e_2, respectively. In total, the proposed hardware performs the encryption in 360 cc.

For the decryption operation, the hardware computes $\mathtt{NTT}(c_1)$ in the same manner as in the encryption. Then, it computes the multiplication $\overline{c_1 s}$, performs $\mathtt{INTT}(\overline{c_1 s})$, and multiplies the result with the powers of Ψ^{-1}. This polynomial multiplication operation is performed in 280 cc. Since decryption operation requires comparison in $\mathbb{Z}_\gamma$, modular addition and modular multiplication in $\mathbb{Z}_t$, as shown in Algorithm 4, the proposed hardware uses additional hardware blocks for these operations. These blocks are not shown in Fig. 2 for simplicity. It should be noted that coefficient-wise modular multiplication operations in $\mathbb{Z}_\gamma$ shown in Step 4 of Algorithm 4 are performed in the multiplier unit by changing modulus from q to γ and require no extra hardware. Finally, the necessary operations are performed as shown in Algorithm 4. The hardware completes one decryption operation in 360 cc.

3) Iterative Hardware: Our iterative hardware utilizes the basic building blocks proposed in our preliminary work [17], in which only NTT operation is implemented in hardware. Remaining operations of encryption and decryption are still realized by SEAL software. In this article, in order to increase the performance of our hardware accelerator, we offload entire `encrypt` and `decrypt` functions onto the FPGA.

a) Iterative NTT unit: For NTT operations, Algorithm 1 is implemented in hardware, and the resulting NTT module is shown in Fig. 4 [17].

b) Overall design: In this article, certain design choices are made to achieve a balanced performance between time and area. For performance reasons, 64 instances of the iterative NU are used in the iterative hardware (see Fig. 5). NTT and INTT are realized in the same hardware, by just changing the precomputed twiddle (ω) factors.

Since the hardware architectures required to realize their operations are similar, we decide to utilize the same NU to perform both NTT and coefficient-wise modular multiplication operations in the NTT domain. The overall design of the iterative hardware architecture is shown in Fig. 5. There are also adder unit performing modular additions and memory for storing intermediate operands as shown in Fig. 5. The iterative hardware uses additional 64 modular multipliers and comparators for implementing modular multiplication and comparison operations in $\mathbb{Z}_\gamma$ shown in Steps 4 and 7 of Algorithm 4, respectively. It also utilizes additional hardware blocks for modular addition and modular multiplication in $\mathbb{Z}_t$ used for decryption as shown in Algorithm 4. These additional hardware blocks are now shown in Fig. 5.

The iterative hardware employs 64 BRAMs for storing twiddle factors. Upper half and lower half of these BRAMs are used for storing the powers of ω and ω^{-1}, respectively. The control logic adjusts the most significant bit of BRAM address for alternating between NTT and INTT. The precomputed parameters $\overline{p_0}$, $\overline{p_1}$, and $\overline{s}$, and the powers of Ψ and Ψ^{-1} are also stored in BRAMs as shown in Fig. 5 and used during encryption and decryption operations.

The iterative hardware performs one NTT and 64 coefficient-wise modular multiplication operations in 80 cc and 8 cc, respectively. Therefore, one polynomial multiplication operation is performed in 192 cc. Also, encryption and decryption operations are completed in 280 cc and 248 cc, respectively.

B. Hardware/Software Codesign Framework

In order to demonstrate that homomorphic encryption/decryption operations of the SEAL library can be accelerated considerably, we designed a proof-of-concept accelerator framework that includes SEAL software and an FPGA accelerator that implements our architectures. For communication between the software stack and FPGA, we utilized reusable integration framework for FPGA accelerators (RIFFA) driver [26], which employs a PCIe connection between CPU and FPGA. Resulting framework is shown in Fig. 6.

In SEAL, there are `encrypt` and `decrypt` functions, which work as described in III. In our modified version of SEAL, `encrypt` and `decrypt` functions send their inputs m, u and c_0, c_1, respectively, to FPGA and once FPGA returns the results to CPU, these functions return them to their caller functions. Precomputed constants such as keys are sent to FPGA only once prior to any invocation of `encrypt` and

`decrypt` functions. In summary, all arithmetic operations in encryption and decryption are performed in FPGA except for sampling of random polynomials and encoding of the plaintext, which are performed in the host CPU and sent to FPGA prior to any operation.

One important aspect of the communication between CPU and FPGA is the utilization of direct memory access (DMA). Instead of bringing data into CPU first, prior to sending it to FPGA, the data are directly sent to FPGA from memory. This way, cache memory is never trashed and running `encrypt` or `decrypt` function does not affect the performance of other operations running on CPU.

To realize our framework, we use the XILINX VC707 Evaluation Kit, which includes a PCIe x8 Gen 2 Connector. XILINX IP Core 7-Series Integrated Block for PCIe provides a 128-bit interface with a 250-MHz clock, which has a 32 Gb/s theoretical maximum bandwidth. As shown in Fig. 6, 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. In [26], it is shown that RIFFA is able to achieve only 76% of the maximum theoretical bandwidth. Therefore, the bandwidth of the PCIe module is assumed to be $\sim$24 Gb/s.

SEAL library uses a 64-bit integer type for storing the coefficients regardless of the size of q. Since we work with 32-bit coefficients, we can pack and send $128/32 = 4$ coefficients per cycle. However, packing four 32-bit coefficients complicates the memory access in SEAL. Therefore, we pack and send $128/64 = 2$ coefficients per cycle. Both encryption and decryption operations take $2 \cdot 1024 = 2048$ coefficients as inputs and CPU can send $(8 \cdot 3 \cdot 10^9)/(8 \cdot 2048) = 183\,105$ encryption or decryption inputs per second with 24-Gb/s bandwidth. In order not to be I/O bounded, the implementation on FPGA must finish its operations in less than $1\ \text{s}/183\,105 = 5.46\ \mu$s. Since the proposed hardware implementations finish the encryption or decryption less than 5.46 μs as demonstrated in Section V, they are not I/O bounded.

V. Results and Comparison

We developed two architectures in Section IV into Verilog modules and realized them using XILINX Vivado 2018.1 tool for the XILINX VC707 Evaluation Kit, utilizing a VIRTEX-7 FPGA (XC7VX485T-2FFG1761C), which has 303600 lookup tables (LUTs), 607200 D flip-flops (DFFs), 2800 DSP48E1s, and 1030 BRAM36E1s. The iterative and four-step hardware use 25.63% and 22.36% of LUTs, 31.6% and 12.52% of RAMB36E1s, and 34% and 21.39% of DSP48E1s in FPGA, respectively.

Many works were reported in the literature proposing hardware accelerators for homomorphic encryption schemes [24], [27]–[39]. Some of these works focus on accelerating the multiplication of two large degree polynomials using NTT-based multiplication schemes [24], [28]–[31], [35], [36]. Other works target accelerating different operations such as full encryption/decryption and homomorphic multiplication operations [27], [32]–[34], [37]. Also, the works in [39] and [38] target fast NTT hardware for lattice-based cryptography, which can also be used for homomorphic encryption schemes. Although our hardware architectures accelerate encryption and decryption operations of the BFV scheme in SEAL, the core part of our architectures is the hardware implementation of a fast polynomial multiplier. For a fair comparison, therefore, we report and compare the hardware and performance results for the polynomial multiplier part of our works and the works in the literature in Table II. We also include the performance results of NTT operation of the works in the literature, if available, in Table II. The proposed iterative and four-step hardware implementations have the lowest latency for both NTT and polynomial multiplication operations compared to works in the literature.

Also, in Table II, we include the implementation results of the iterative hardware on a low-cost Spartan-6 FPGA board [17]. The results show that the timing result is comparable to the one in [29]. Note that we achieve a comparable timing result using a general ciphertext modulus q, while [29] uses a special modulus. In terms of area, our design uses much less distributed logic at the expense of ten additional DSPs.

Although there are other accelerators [40] in the literature performing RLWE encryption and decryption, these works use small parameters and not designed for homomorphic operations. Thus, they are not included in the comparison.

Although the proposed work has relatively small parameters for homomorphic operations and has a multiplicative depth of 1, it can be extended to a new design with larger ring degree and ciphertext modulus using exactly the same arithmetic units in this article. For example, for a design with a ring degree of 4096 and 180-bit ciphertext modulus, we just need to update the control unit of NTT hardware so that it can work for ring degree of 4096 instead of 1024 using exactly the same NUs. Also, the ciphertext modulus can be increased to 180-bit by using exactly the same polynomial multipliers in this article with additional Chinese remainder theorem (CRT) [41] operations employing CRT. In such setting, the proposed hardware needs a CRT unit that transforms each 180-bit coefficient into six coefficients in six 32-bit primes, performs operations separately for each 32-bit prime using desired number of hardware units in parallel, and converts coefficients in six 32-bit primes into 180-bit coefficients. Therefore, the arithmetic blocks proposed in this article with small parameter set can be used to design high-performance hardware for larger parameter sets with minor modifications. We present two different scaled versions of the proposed architectures for $n = 4096$ with 32-bit q and $n = 4096$ with 180-bit q and reported estimated timing and area results, showing that timing and area results are linearly proportional to n and q, in Table II.

Software implementation using only SEAL completes encryption, decryption, and one polynomial multiplication in 151, 65.7, and 28.8 μs, respectively. Our FPGA implementation of the iterative hardware, excluding I/O operations, performs encryption, decryption, and polynomial multiplication in 1.4, 1.24, and 0.96 μs, respectively, resulting in 108×, 53×, and 30× speedup values for those operations when compared with the pure software implementation. Similarly, the FPGA implementation of the four-step hardware performs both encryption and decryption operations in 1.8 μs and polynomial multiplication in 1.4 μs, resulting in 84×, 37×, and 21×

TABLE II
COMPARATIVE TABLE

Work	Scheme	Platform	n	q	LUT / DSP / BRAM	Clock (MHz)	Latency (ms)	
							NTT	Pol. Mul.
[33]	FV	VIRTEX-6	65536	30-bit	72K / 250 / 106	100	–	3.376
[34]	YASHE	VIRTEX-7	4096	125-bit	69K / 144 / –	100	–	1.960
[27]	FV	STRATIX-V	2560	125-bit	30K / 100 / –	331	–	0.583
[32]	FV	Zynq UltraScale	4096	30-bit	64K / 200 / 400	225	0.073	0.171
[24]	LTV	VIRTEX-7	32768	32-bit	219K / 768 / 193	250	0.051	0.152
[31][a]	HE	SPARTAN-6	1024	30-bit	1644 / 1 / 6.5	200	–	0.110
[30][a]	HE	SPARTAN-6	1024	17-bit	– / 3 / 2	–	–	0.100
[29][a]	RLWE	SPARTAN-6	256	21-bit	2829 / 4 / 4	247	–	0.006
	SHE	SPARTAN-6	1024	31-bit	6689 / 4 / 8	241	–	0.033
[28]	RLWE	VIRTEX-7	4096	30-bit	54K / 517 / 208	200	–	0.010
[35][a]	RLWE	SPARTAN-6	256	21-bit	14K / 128 / 1	233	–	0.00094
			512	23-bit	18K / 128 / 2.5	200	–	0.00177
[36][a]	RLWE	KINTEX-7	256	17-bit	317 / 1 / –	333	0.102	–
[37][b]	RLWE	40nm CMOS	256	24-bit	106K / – / –	72	0.017	–
[38][b]	RLWE	40nm CMOS	512	18-bit	– / – / –	300	0.0016	–
[39][b]	RLWE	UMC 65nm	1024	17-bit	14K / – / –	25	0.041	–
Iterative [17]	FV	SPARTAN-6	1024	32-bit	1208 / 14 / 14	212	–	0.037
Iterative	FV	VIRTEX-7	1024	32-bit	77K / 952 / 325.5	200	0.0004	0.00096
Four-Step					67K / 599 / 129		0.0007	0.00140
Iterative[c]	FV	VIRTEX-7	4096	32-bit	∼80K / 952 / 325.5	∼200	∼0.00175	∼0.00420
Four-Step[c]					∼70K / 599 / 129		∼0.0023	∼0.00475
Iterative[d]	FV	VIRTEX-7	4096	180-bit	∼160K / 1904 / 651	∼200	∼0.00525	∼0.01260
Four-Step[d]					∼140K / 1198 / 258		∼0.0069	∼0.01425

[a]:Fixed q. [b]:Multiple n and q. [c]:Scaled for n=4096. [d]:Scaled for n=4096 and q=180-bit (assuming two 32-bit hardware are instantiated, excluding CRT).

speedup values for encryption, decryption, and one polynomial multiplication, respectively. The iterative hardware performs both encryption and decryption faster than the four-step hardware at the expense of more resources.

Transmission of a polynomial of degree 1024 with 32-bit coefficients between CPU and FPGA via DMA takes 2.73 μs by packing two coefficients per cycle. For iterative hardware, for encryption operation, without pipelining of the transmission and the FPGA computation and with half-duplex PCIe communication, we achieve $5.46 + 1.4 + 5.46 = 12.32$-$\mu$s latency, where 5.46 μs is spent for sending the input, 1.4 μs for the encryption operation, and another 5.46 μs is spent for receiving the output. In comparison with pure software implementation, this indicates a 12× speedup for encryption. Similarly, for decryption operation, we obtain $5.46 + 1.24 + 2.73 = 9.42$-$\mu$s latency, which is a 7× speedup over the software implementation. Also, we achieve a throughput of almost 81K and 106K for encryption and decryption operations, respectively, per second without pipelining. Performance results for the four-step hardware can be calculated similarly.

In current implementation, PCIe works in the half-duplex mode, where the host CPU can either send or receive one encryption/decryption operation at a time over PCIe. If we use PCIe in the full-duplex mode where PCIe can send and receive data at the same time and overlap I/O operations over PCIe with actual encryption and decryption operations, as shown in Table III, the proposed hardware can send the result of one encryption or decryption operation back to the host CPU in $\max(5.46, 1.4, 1.24, 2.73) = 5.46$ μs after filling the pipeline. In this setting, the proposed framework can perform $1/5.46\mu s = 183150$ encryption or decryption operations per second. Compared to $1/151\mu s = 6622$ encryption and $1/65.7\mu s = 15220$ decryption operations per second, we can achieve 27× and 12× speedup over pure software encryption and decryption implementations, respectively.

TABLE III
PIPELINING OF I/O OPERATIONS OVER PCIe

Time (μs)	Input	Operation	Output
0	Enc1	–	–
5.46	Dec1	Enc1	–
10.92	Enc2	Dec1	Enc1
16.38	...	Enc2	Dec1
21.84	...	...	Enc2
...	...	...	...

VI. CONCLUSION

We presented FPGA implementations of two fast and highly parallelized hardware architectures for the encryption and decryption operations of the BFV homomorphic encryption scheme. We utilized our architectures in an accelerator framework for the encryption and decryption operations of the BFV homomorphic encryption scheme implemented in the SEAL. We adopt a hardware/software codesign approach, in which encryption and decryption operations are offloaded to an FPGA, while the rest of operations in the BFV scheme of SEAL are executed in software running on a desktop computer. We realized the framework on an FPGA connected to the PCIe bus of an off-the-shelf desktop computer. We used a XILINX VC707 board for our implementation. We improved the latency of the encryption and decryption by almost 12× and 7×, respectively, compared to their pure software implementations.

Also, we show that utilizing efficient FPGA accelerators for homomorphic encryption libraries such as SEAL is very

promising. Our results show that this article can be extended for accelerating homomorphic operations such as homomorphic multiplication when high performance is required from both latency and throughput perspectives. As future work, we target accelerating homomorphic multiplication, which requires acceleration for high performance, using our accelerator framework. Our results indicate that our framework will result in potentially much higher speedup values for homomorphic operations as they require much less communication between FPGA and CPU and the transmission can be completely overlapped by computations in FPGA.

Finally, with small modifications, the core arithmetic units in our accelerator can be used to implement ring arithmetic with larger ring degrees and modulus sizes. Currently, we are working on such new design based on our current architecture and framework to accelerate more involved homomorphic operations, and the results will be presented in our future work.

REFERENCES

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations of Secure Computation*. New York, NY, USA: Academic, 1978, pp. 169–179.

[2] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2009.

[3] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. STOC*, New York, USA, May 2012, pp. 1219–1234.

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptol. ePrint Arch., Tech. Rep. 2012/144, 2012.

[5] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, no. 1, pp. 1–33, Jan. 2010.

[6] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *Proc. 16th Int. Workshop Cryptogr. Hardw. Embedded Syst. (CHES)*, Busan, South Korea, Sep. 2014, pp. 371–391.

[7] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede, "Efficient software implementation of ring-LWE encryption," in *Proc. DATE*, Mar. 2015, pp. 339–344.

[8] S. Halevi and V. Shoup, "Algorithms in HElib," in *Proc. Annu. Int. Cryptol. (CRYPTO)*. Santa Barbara, CA, USA: Springer-Verlag, Aug. 2014, pp. 554–571.

[9] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "NFL$_{LIB}$: NTT-based fast lattice library," in *Topics in Cryptology—CT-RSA 2016*. San Francisco, CA, USA: Springer, 2016, pp. 341–356.

[10] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans*. Koper, Slovenia: Springer, Sep. 2016, pp. 169–186.

[11] Microsoft Research, Redmond, WA, USA. (Feb. 2019). *Microsoft SEAL (release 3.2)*. [Online]. Available: https://github.com/Microsoft/SEAL

[12] J. Wang, A. Arriaga, Q. Tang, and P. Y. A. Ryan, "CryptoRec: Privacy-preserving recommendation as a service," 2018, *arXiv:1802.02432*. [Online]. Available: https://arxiv.org/abs/1802.02432

[13] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *Trans. CHES*, vol. 2018, no. 2, pp. 70–95, 2018.

[14] S. Angel, H. Chen, K. Laine, and S. Setty, "PIR with compressed queries and amortized query processing," Cryptol. ePrint Arch., Tech. Rep. 2017/1142, 2017.

[15] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Cryptology and Network Security*. Milan, Italy: Springer, Nov. 2016, pp. 124–139.

[16] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[17] A. C. Mert, E. Ozturk, and E. Savas, "Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture," Cryptol. ePrint Arch., Tech. Rep. 2019/109, 2019.

[18] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Advances in Cryptology—CRYPTO 2012*. Berlin, Germany: Springer, 2012, pp. 868–886.

[19] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology—EUROCRYPT 2010*. French Riviera, France, 2010, pp. 1–23.

[20] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proc. 37th Annu. ACM STOC*, Baltimore, MD, USA, 2005, pp. 84–93.

[21] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Selected Areas in Cryptography—SAC 2016*. St. John's, NL, Canada: Springer, 2016, pp. 423–442.

[22] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[23] T. Yanık, E. Savaş, and Ç. K. Koç, "Incomplete reduction in modular arithmetic," *IEE Proc.-Comput. Digit. Techn.*, vol. 149, no. 2, pp. 46–52, Mar. 2002.

[24] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 3–16, Jan. 2017.

[25] E. Kalali, A. C. Mert, and I. Hamzaoglu, "A computation and energy reduction technique for HEVC discrete cosine transform," *IEEE Trans. Consum. Electron.*, vol. 62, no. 2, pp. 166–174, May 2016.

[26] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A reusable integration framework for FPGA accelerators," in *Proc. IEEE 20th Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2012, pp. 216–219.

[27] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm," *IEEE Trans. Comput.*, vol. 67, no. 3, pp. 335–347, Mar. 2018.

[28] J. Cathébras, A. Carbon, P. Milder, R. Sirdey, and N. Ventroux, "Data flow oriented hardware design of RNS-based polynomial multiplication for SHE acceleration," *IACR Trans. CHES*, vol. 2018, no. 3, pp. 69–88, Aug. 2018.

[29] D. D. Chen *et al.*, "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 62, no. 1, pp. 157–166, Jan. 2015.

[30] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient FPGA implementations of lattice-based cryptography," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2013, pp. 81–86.

[31] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology—LATINCRYPT 2012*. Berlin, Germany: Springer, 2012, pp. 139–158.

[32] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," Cryptol. ePrint Arch., Tech. Rep. 2019/160, 2019.

[33] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.

[34] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Proc. CHES*, Saint-Malo, France, Sep. 2015, pp. 143–163.

[35] X. Feng, S. Li, and S. Xu, "RLWE-oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm," *IEEE Trans. Circuits Syst., II, Exp. Briefs*, to be published.

[36] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, to be published.

[37] U. Banerjee, A. Pathak, and A. P. Chandrakasan, "2.3 An energy-efficient configurable lattice cryptography processor for the quantum-secure Internet of Things," in *Proc. IEEE ISSCC*, Feb. 2019, pp. 46–48.

[38] S. Song, W. Tang, T. Chen, and Z. Zhang, "Leia: A 2.05 mm^2 140 mW lattice encryption instruction accelerator in 40nm CMOS," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr. 2018, pp. 1–4.

[39] T. Fritzmann and J. Sepúlveda, "Efficient and flexible low-power NTT for lattice-based cryptography," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2019, pp. 141–150.

[40] G. Seiler, "Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography," in *Proc. IACR Cryptol. ePrint Arch.*, vol. 2018, p. 39, 2018.

[41] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," *Notices AMS*, vol. 46, no. 2, pp. 203–213, 1999.