Microprocessors and Microsystems 78 (2020) 103219

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

# FPGA implementation of a run-time configurable NTT-based polynomial multiplication hardware

Ahmet Can Mert *, Erdinç Öztürk, Erkay Savaş

*Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey*

ARTICLE INFO

*Keywords:*
Number theoretic transform
Polynomial multiplication
Modular multiplier
SEAL

ABSTRACT

Multiplication of polynomials of large degrees is the predominant operation in lattice-based cryptosystems in terms of execution time. This motivates the study of its fast and efficient implementations in hardware. Also, applications such as those using homomorphic encryption need to operate with polynomials of different parameter sets. This calls for design of configurable hardware architectures that can support multiplication of polynomials of various degrees and coefficient sizes.

In this work, we present the design and an FPGA implementation of a *run-time configurable* and highly parallelized NTT-based polynomial multiplication architecture, which proves to be effective as an accelerator for lattice-based cryptosystems. The proposed polynomial multiplier can also be used to perform Number Theoretic Transform (NTT) and Inverse NTT (INTT) operations. It supports 6 different parameter sets, which are used in lattice-based homomorphic encryption and/or post-quantum cryptosystems. We also present a hardware/software co-design framework, which provides high-speed communication between the CPU and the FPGA connected by PCIe standard interface provided by the RIFFA driver [1]. For proof of concept, the proposed polynomial multiplier is deployed in this framework to accelerate the decryption operation of Brakerski/Fan-Vercauteren (BFV) homomorphic encryption scheme implemented in Simple Encrypted Arithmetic Library (SEAL), by the Cryptography Research Group at Microsoft Research [2]. In the proposed framework, polynomial multiplication operation in the decryption of the BFV scheme is offloaded to the accelerator in the FPGA via PCIe bus while the rest of operations in the decryption are executed in software running on an off-the-shelf desktop computer. The hardware part of the proposed framework targets Xilinx Virtex-7 FPGA device and the proposed framework achieves the speedup of almost 7 × in latency for the offloaded operations compared to their pure software implementations, excluding I/O overhead.

## 1. Introduction

Polynomial multiplication operation is excessively used in lattice-based cryptosystems, such as post-quantum key-exchange/signature protocols [3–5] and homomorphic encryption applications [6,7]. Being the major operation of the lattice-based cryptosystems, it also creates the computational bottleneck in their implementations. Therefore, it has been recently studied in many works in the literature. The Number Theoretic Transform (NTT)-based polynomial multiplication algorithm is a frequently used technique to improve the performance of polynomial multiplication operation for especially large polynomial degrees and it is widely employed in lattice-based cryptosystems.

The design of NTT-based polynomial multiplication operation is mainly based on two parameters: the degree of the cyclotomic polynomial of the polynomial ring, $n$, and the bit length of the coefficient modulus, $K = \lceil \log q \rceil$., where $q$ is the coefficient modulus. For cryptographic applications utilizing such different parameters, separate polynomial multipliers need to be designed and implemented. For example, post-quantum key-exchange protocol `CRYSTALS-Kyber (v1)` [3] uses parameters $n$=256 and $K$=13, where $q$=7681 is a 13-bit prime, while Simple Encrypted Arithmetic Library (SEAL) [2] homomorphic encryption library by Microsoft uses parameters $n$ ranging from 1024 to 32,768 and $K$ ranging from 14-bit to 60-bit. Therefore, this is the motivation for a configurable NTT-based polynomial multiplier architecture, which supports multiple parameter sets and applications instead of separate architectures for each application with fixed parameters.

To this end, in this work, we propose a *run-time configurable* and highly parallelized NTT-based polynomial multiplier architecture for

* Corresponding author.
*E-mail addresses:* ahmetcanmert@sabanciuniv.edu (A.C. Mert), erkays@sabanciuniv.edu (E. Savaş).

https://doi.org/10.1016/j.micpro.2020.103219
Received 6 December 2019; Received in revised form 28 May 2020; Accepted 3 August 2020
Available online 7 August 2020

hardware realizations. Our motivation in our study is two fold: *i*) our architecture effectively supports different parameter sets with high efficiency and *ii*) thus aims to improve the performance of a wide range of lattice-based cryptosystems.

Here, we also propose a hardware/software co-design framework, which provides a high-speed communication between the CPU and the FPGA by utilizing the RIFFA driver [1] employing a PCIe standard interface. For proof of concept, our NTT-based polynomial multiplier architecture is utilized in this framework to accelerate the decryption operation of Brakerski/Fan-Vercauteren (BFV) homomorphic encryption scheme implemented in the SEAL [2]. In the proposed framework, polynomial multiplication operation in the decryption of the BFV scheme is offloaded to the accelerator in the FPGA via PCIe bus while the rest of operations in the decryption of the BFV scheme are executed in software running on an off-the-shelf desktop computer. Offloading the computation to the 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) overhead. This overhead can be prohibitively high if the nature and cost of the offloading are not factored in the accelerator design.

To address the speed and configurability requirements, three crucial design goals are considered in this work: *i*) hardware accelerator architecture should be designed to provide significant levels of speedup over software implementations, *ii*) the overhead due to communication between hardware and software components should be taken into account as a design parameter or constraint and *iii*) a balanced implementation in terms of area and throughput should be designed as the proposed architecture supports different parameter sets and aims providing acceleration for a variety of applications. Most works in the literature focus solely on the first goal and report no accurate speedup values subsuming the I/O overhead. 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.

To summarize, our contributions in this paper are listed as follows:

- We present a novel architecture for a run-time configurable and highly parallelized NTT-based polynomial multiplier and its FPGA implementation. Besides polynomial multiplication, the proposed architecture can perform NTT and Inverse NTT (INTT) operations. The proposed architecture supports 6 parameter sets $(n, K) = \{(256, 16), (512,16), (1024,16), (1024,32), (2048,32), (4096, 32)\}$, which are utilized in the lattice-based cryptosystems, with a flexible memory addressing scheme. This work extends our earlier work [8] implementing an NTT-based polynomial multiplier architecture for a fixed parameter set, namely (1024,32), which supports limited number of applications. Moreover, the proposed architecture in this work can perform polynomial multiplication in two different ways while our earlier work [8] can perform only one type of polynomial multiplication.
- We introduce several optimizations for NTT operation. We also slightly modify the NTT operation loops in order to be able to efficiently parallelize NTT computations [8]. 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 implementation supports multiple parameter sets and aims accelerating polynomial multiplication operation in different applications, we follow a balanced design approach in terms of area and throughput. In addition to these, since our implementation is targeting cryptographic applications, for security-aware realizations, the proposed polynomial multiplier architecture is designed to run in constant time for every possible input combination for a parameter set.
- We introduce a novel configurable word-level Montgomery modular multiplier architecture for any NTT-friendly modulus, which provides comparable time performance to those using special modulus. The modular multiplier proposed in this work extends the modular multiplier architecture proposed in our earlier work [8], which implements modular multiplier architecture for a fixed parameter set, namely (1024,32).
- We propose a hardware/software co-design 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 $\sim$ 24 Gbps half-duplex I/O speed. For proof of concept, we accelerate polynomial multiplication utilized in the decryption operation implemented in the SEAL. Every time a decryption function is invoked in the SEAL, the polynomial multiplication operation is offloaded to the FPGA device via the fast PCIe connection. As a case study, we selected three parameter sets (1024,14), (1024,27), (2048,29) from the SEAL. With our approach and the selected parameter sets, the latency of polynomial multiplication is improved by up to 7 $\times$ with about a 9.2% utilization of the Virtex-7 resources. The proposed framework in this work extends the framework in our earlier work [8], which is configured to accelerate encryption operation implemented in the SEAL and works for only a single parameter set, namely (1024,32). As the accelerator framework provides a simple interface and the proposed architecture supports a range of parameter sets for NTT, INTT and polynomial multiplication operations, it can be configured for use with other FHE libraries relying on ring learning with errors security assumption or lattice-based cryptosystems.

## 2. Background

In this section, we give the definition of arithmetic operations highly utilized in the lattice-based cryptosystems.

**Input:** $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} \in \mathbb{Z}_q[x]/(x^n + 1)$
**Input:** $B(x) = b_0 + b_1 x + \ldots + b_{n-1}x^{n-1} \in \mathbb{Z}_q[x]/(x^n + 1)$
**Input:** primitive $2n$th root of unity $\Psi \in \mathbb{Z}_q$
**Output:** $C(x) = c_0 + c_1 x + \ldots + c_{n-1}x^{n-1} \in \mathbb{Z}_q[x]/(x^n + 1)$
1: $\hat{A}(x) = (a_0, a_1, \ldots, a_{n-1}) \odot (1, \Psi^1, \Psi^2, \ldots, \Psi^{n-1})$ ▷ Pre-processing
2: $\hat{B}(x) = (b_0, b_1, \ldots, b_{n-1}) \odot (1, \Psi^1, \Psi^2, \ldots, \Psi^{n-1})$ ▷ Pre-processing
3: $\mathrm{NTT}_{\hat{A}} = \mathtt{NTT}_n(\hat{A}(x))$
4: $\mathrm{NTT}_{\hat{B}} = \mathtt{NTT}_n(\hat{B}(x))$
5: $\mathrm{NTT}_{\hat{C}} = \mathrm{NTT}_{\hat{A}} \odot \mathrm{NTT}_{\hat{B}}$
6: $\hat{C}(x) = \mathtt{INTT}_n(\mathrm{NTT}_{\hat{C}})$
7: $C(x) = (\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}) \odot (1, \Psi^{-1}, \Psi^{-2}, \ldots, \Psi^{-(n-1)})$ ▷ Post-processing
8: **return** $C(x)$

**Algorithm 1.** NTT-based polynomial multiplication with negative wrapped convolution technique.

### 2.1. NTT-based polynomial multiplication

A fundamental arithmetic and time-consuming operation in homomorphic encryption and lattice-based cryptography is the multiplication of two large degree polynomials. This operation is defined over the ring of polynomials $\mathbb{Z}_q[x]/\phi(x)$, i.e., polynomials of degree $(n-1)$ with coefficient modulo $q$ for $\phi(x) = (x^n + 1)$. Therefore, the polynomial multiplication operation in $\mathbb{Z}_q[x]/(x^n+1)$ takes polynomials $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ as inputs, and returns the output polynomial $C(x) = \sum_{i=0}^{n-1} c_i x^i$.

Schoolbook polynomial multiplication technique has $\mathcal{O}(n^2)$ complexity and results in very inefficient software and hardware implementations for very large $n$ values. NTT-based polynomial multiplication reduces complexity to $\mathcal{O}(n \cdot \log n)$ and provides efficient algorithms for the multiplication of two polynomials. It utilizes NTT/INTT operations to convert polynomial multiplication operation into coefficient-wise multiplication operations.

Although NTT-based polynomial multiplication reduces the complexity of the polynomial multiplication operation, the resulting polynomial, $C(x)$, still needs to be reduced with the reduction polynomial, $\phi(x)$, after the multiplication operation. This reduction operation could be avoided using a technique called *negative wrapped convolution* when $\phi(x) = (x^n+1)$ and $q \equiv 1 \pmod{2n}$. This technique eliminates reduction operation by introducing pre-processing and post-processing operations for input and output polynomials of polynomial multiplication operation respectively.

In pre-processing and post-processing operations, the coefficients of the input and output polynomials are multiplied with the powers of $\Psi \in \mathbb{Z}_q$ and $\Psi^{-1} \in \mathbb{Z}_q$ respectively, where $\Psi$ is the primitive $2n$th root of unity satisfying $\Psi^{2n} \equiv 1 \pmod{q}$ and $\Psi^i \neq 1 \pmod{q}$ $\forall i < 2n$, where $q \equiv 1 \pmod{2n}$. NTT-based polynomial multiplication operation with *negative wrapped convolution* technique is shown in Algorithm 1, where $\odot$ represents coefficient-wise multiplication and $\texttt{NTT}_n$ and $\texttt{INTT}_n$ represent $n$-point (pt) NTT and INTT operations respectively.

### 2.2. Number theoretic transform

NTT is a discrete Fourier transform defined over the ring $\mathbb{Z}_q/\phi_m(x)$, where $\phi_m(x)$ is the $m$th cyclotomic polynomial and $m = 2 \cdot n$. The $n$-pt NTT operation transforms an $(n-1)$ degree polynomial, $A(x) = \sum_{i=0}^{n-1} a_i x^i$, in polynomial domain to an $(n-1)$ degree polynomial, $\overline{A}(x) = \sum_{i=0}^{n-1} \mathcal{A}_i x^i$, in NTT domain, where each NTT coefficient $\mathcal{A}_i$ is defined as $\mathcal{A}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ over $\mathbb{Z}_q$ for $i = 0,1,\ldots,n-1$. INTT can be similarly calculated as $a_i = n^{-1} \sum_{j=0}^{n-1} \mathcal{A}_j \omega^{-ij}$ in $\mathbb{Z}_q$ for $i = 0,1,\ldots,n-1$. The NTT operation uses the constant, $\omega \in \mathbb{Z}_q$, called twiddle factor, which is $n$th root of unity. The twiddle factor should satisfy the conditions $\omega^n \equiv 1 \pmod{q}$ and $\omega^i \neq 1 \pmod{q}$ $\forall i < n$, where $q \equiv 1 \pmod{n}$.

There are many NTT algorithms present in the literature [9–13]. In this work, we utilize the iterative version of in-place *decimation-in-frequency (DIF)* NTT and INTT algorithms which utilize Gentleman-Sande butterfly [9]. The NTT and INTT schemes are shown in Algorithm 2 and Algorithm 3 respectively. The NTT operation takes input polynomial in standard order and produces output polynomial in bit-reversed order while the INTT operation takes input polynomial in bit-reversed order and produces output polynomial in standard order. Therefore, any need for bit-reversal operation is avoided during polynomial multiplication operation. The $\texttt{br}(k, l-1)$ operation in Algorithm 3 performs bit-reversal on $(l-1)$-bit integer $k$, where $l = \log n$. INTT operation also requires the coefficients of output polynomial to be multiplied with $n^{-1} \pmod{q}$ as shown in steps 19–21 in Algorithm 3.

## 3. The proposed architecture

In this section, the proposed run-time configurable polynomial multiplier architecture, its main building blocks, the design techniques we used for our entire framework and our optimizations are explained.

### 3.1. The configurable word-level montgomery modular multiplier

The proposed configurable modular multiplier hardware consists of two blocks, an integer multiplier hardware and a configurable word-level Montgomery modular reduction hardware. It supports modular multiplication operation only with NTT-friendly modulus, which satisfies $q \equiv 1 \pmod{2n}$, for $K$ ranging from 10 to 32.

First, we design a 32-bit integer multiplier. The proposed integer multiplier hardware utilizes four DSP blocks in FPGAs and an adder tree structure. Since we target FPGA architecture with 18-bit × 25-bit signed core multipliers in DSP blocks, we divide each 32-bit integer inputs into 16-bit chunks and perform four 16-bit × 16-bit multiplications using four DSP blocks in the FPGA for generating partial products. Then, the calculated partial products are added up using carry save adders (CSA) to generate the result of multiplication. The designed 32-bit integer multiplier hardware is shown in Fig. 1. The proposed 32-bit integer multiplier hardware performs a multiplication operation in 2 clock cycles. It is pipelined and uses optional output registers of DSP blocks as pipeline registers, which eliminates the need to utilize FPGA fabric registers for pipelining. Therefore, the proposed integer multiplier can produce one multiplication result per clock cycles after filling the pipeline. The proposed integer multiplier works for any integer inputs with 32 or less bits.

After the multiplication operation, the result needs to be reduced back to the bit-length of the modulus. For a configurable architecture, we use the modified word-level Montgomery modular reduction algorithm proposed in [8] as shown in Algorithm 4. The modular reduction architecture implemented in [8] works for a fixed parameter set, $n$=1024 and $K$=32 while we, in this work, implement a configurable architecture, which supports multiple parameter sets.

The proposed word-level Montgomery modular reduction algorithm uses the property of NTT-friendly modulus with *negative wrapped convolution* technique, $q \equiv 1 \pmod{2n}$, and it divides Montgomery reduction operation into smaller steps. Any NTT-friendly modulus $q$ with this property can be written as $q = q_H \cdot 2^w + 1$, where $w = \log_2(2n)$ is being the *word size*. For example, for parameters $n$=256 and $K$=16, $q$ can be written as $q_H \cdot 2^9 + 1$ with $w$=9. Due to $q = q_H \cdot 2^w + 1$, Montgomery constant $\mu = -q^{-1} \pmod{2^w}$ becomes $-1 \pmod{2^w}$. Then, the multiplication operation $A \cdot B \cdot (-q^{-1}) \pmod{2^w}$ in the Montgomery algorithm becomes $-A \cdot B \pmod{2^w}$. This converts one multiplication operation in the Montgomery algorithm into the two's complement operation as shown in step 5 of the Algorithm 4. The word-level Montgomery algorithm requires $L = \left\lceil \frac{K}{w} \right\rceil$ iterations for a $K$-bit modulus. For example, for parameters $n$=256 and $K$=16, $L = \left\lceil \frac{16}{9} \right\rceil = 2$ iterations are required.

The modular multiplier architecture in this work can be configured to perform modular multiplication operation for different parameter sets. The proposed configurable word-level Montgomery modular multiplier architecture is shown in Fig. 2. Since maximum iteration count, $L$, is $\left\lceil \frac{32}{13} \right\rceil = 3$, the proposed modular multiplier uses three units performing $X \cdot Y + Z + carry$ operation. For parameters (256,16), (512,16) and (1024,16), which require $L$=2 iterations, the last $X \cdot Y + Z + carry$ operation is eliminated by selecting $X$, $Y$, $Z$ and $carry$ inputs of the third unit as 1, $T$3, 0 and 0, respectively.

$X \cdot Y + Z + carry$ operation is a multiply-accumulate operation, which can be realized using one DSP block inside the FPGAs. Therefore, in the

**Input:** $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$ in standard order
**Input:** primitive $n$th root of unity $\omega \in \mathbb{Z}_q$, $n = 2^l$
**Output:** $\bar{a}(x) \in \mathbb{Z}_q[x]/(x^n+1)$ in bit-reversed order

**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**
      $U \leftarrow a[2 \cdot j \cdot m + k]$
      $V \leftarrow a[2 \cdot j \cdot m + k + m]$
      $N0 \leftarrow (U+V) \pmod q$
      $N1 \leftarrow (U-V) \cdot \omega^{2^{i-1}k} \cdot \pmod q$
      $a[2 \cdot j \cdot m + k] \leftarrow N0$
      $a[2 \cdot j \cdot m + k + m] \leftarrow N1$
    **end for**
  **end for**
**end for**
**return** $a$

**Algorithm 2.** Iterative version of in-place DIF NTT [9].

proposed modular multiplier architecture, each $X \cdot Y + Z + carry$ operation is implemented using one DSP block. For a given parameter set, the proposed modular multiplier selects proper inputs for DSP blocks and performs the modular multiplication operation in constant-time. For example, for parameters $n$=256 and $K$=16, the proposed architecture selects $T1$, $T2$ and $q$ shifted by 9 for the first two DSP blocks and eliminates the last $X \cdot Y + Z + carry$ operation as explained before. For given $w$ and $L$, the proposed modular reduction architecture supports modulus with $w \cdot (L-1) < K \leq w \cdot L$. Therefore, for a parameter set, the proposed architecture supports a range of $K$ instead of single $K$ value. For example, for (1024,32) with $w$=11 and $L$=3, the proposed architecture supports modulus with $22 < K \leq 33$. Supported $K$ range for each parameter set is listed in Table 1. As shown in the table, the proposed architecture can support modulus up to 39-bit long for parameter set (4096,32). Therefore, although the proposed polynomial multiplier architecture supports modulus up to 32-bit, it can be easily extended to 39-bit with slight modification of integer multiplier.

The word-level Montgomery modular reduction algorithm takes $A \cdot B$ as input and produces $A \cdot B \cdot R^{-1} \pmod q$, where $R = 2^{w \cdot L}$. Therefore, the output of the word-level Montgomery modular reduction algorithm should be multiplied with $R$ for eliminating extra $R^{-1}$ in the result. In this work, this extra multiplication operation is avoided by multiplying one of the multiplication inputs with $R$ or the power of $R$.

The proposed modular multiplier architecture performs a modular multiplication operation in 6 clock cycles for all parameter sets. It is pipelined and uses internal registers of DSP blocks as pipeline registers. Therefore, the proposed configurable modular multiplier can produce one multiplication result per clock cycles after filling the pipeline.

Compared to the conventional Montgomery and Barrett algorithms,

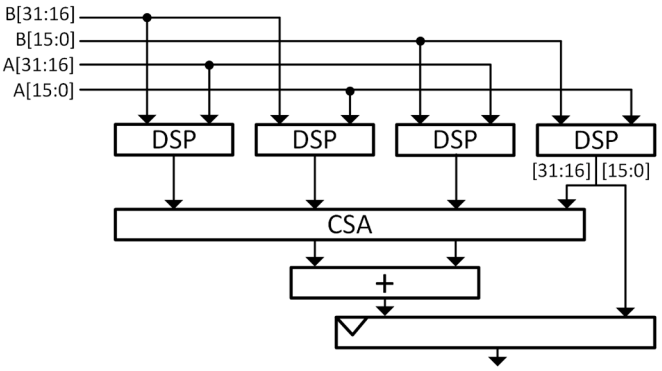



**Fig. 1.** 32-bit integer multiplier.

**Input:** $\bar{a}(x) \in \mathbb{Z}_q[x]/(x^n+1)$ in bit-reversed order
**Input:** modular inverse of primitive $n$th root of unity $\omega^{-1} \in \mathbb{Z}_q$, $n = 2^l$
**Output:** $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$ in standard order

$m = 1$
$v = n$
**while** $v > 1$ **do**
  **for** $i$ from 0 by 1 to $(m-1)$ **do**
    $k = 0$
    **for** $j$ from $i$ by $2 \cdot m$ to $(n-2)$ **do**
      $U \leftarrow a[j]$
      $V \leftarrow a[j+m]$
      $N0 \leftarrow (U+V) \pmod q$
      $N1 \leftarrow (U-V) \cdot \omega^{-\mathrm{br}(k,l-1)} \pmod q$
      $a[j] \leftarrow N0$
      $a[j+m] \leftarrow N1$
      $k = k+1$
    **end for**
  **end for**
  $m = 2 \cdot m$
  $v = v/2$
**end while**
**for** $i$ from 0 by 1 to $(n-1)$ **do**
  $a[i] \leftarrow a[i] \cdot (1/n) \pmod q$
**end for**
**return** $a$

**Algorithm 3.** Iterative version of in-place DIF INTT [9].

**Input:** $C = A \cdot B$ (a $2K$-bit positive integer)
**Input:** $w = \log_2(2n)$ (word size)
**Input:** $L = \lceil \frac{K}{w} \rceil$ (iteration count)
**Input:** $q$ (a $K$-bit positive integer, $q = q_H \cdot 2^w + 1$)
**Output:** $Res = C \cdot R^{-1} \pmod q$ where $R = 2^{w \cdot L} \pmod q$

1: $T1 = C$
2: **for** $i$ from 0 to $(L-1)$ **do**
3: $T1_H = T1 >> w$
4: $T1_L = T1 \pmod{2^w}$
5: $T2 = 2'scomplementofT1_L$
6: $carry = T2[w-1] \vee T1_L[w-1]$
7: $T1 = T1_H + (q_H \cdot T2[w-1:0]) + carry$
8: **end for**
9: $T4 = T1 - q$
10: **if** $(T4 < 0)$ **then** $Res = T1$ **else** $Res = T4$
11: **return** $Res$

**Algorithm 4.** Word-level montgomery reduction algorithm modified for NTT-friendly modulus [8].

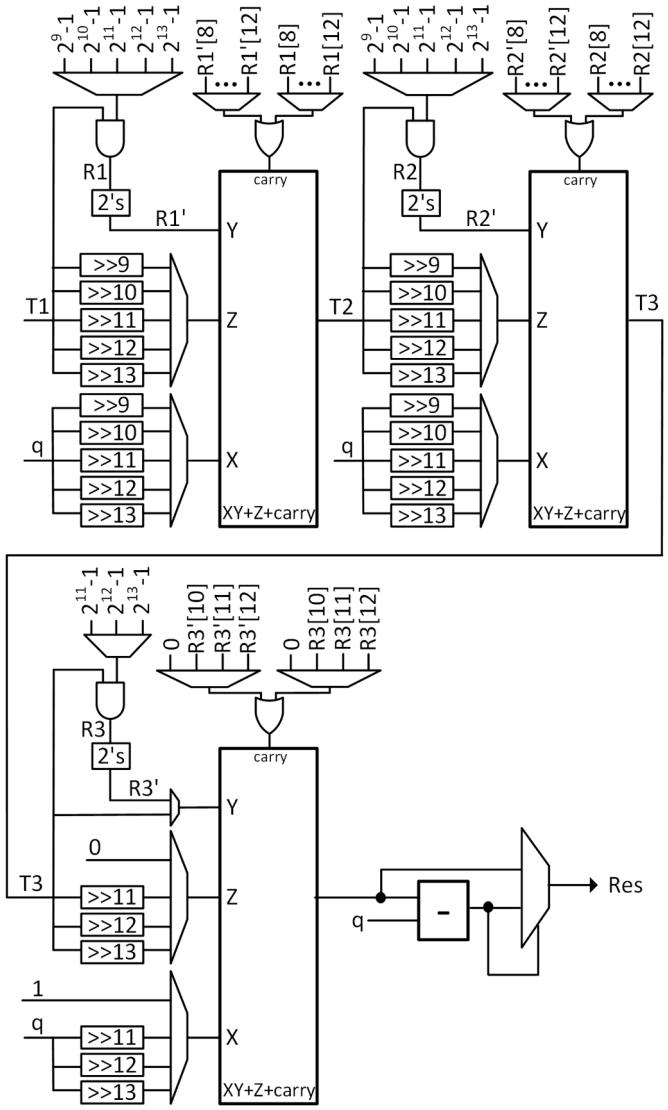



**Fig. 2.** Configurable word-level montgomery modular reduction hardware.

the proposed word-level Montgomery algorithm reduces the size of multiplication operations and provides better utilization for FPGA implementations. For example, for the parameter set (1024,32), the conventional Montgomery algorithm uses two 32 × 32 multipliers, which require 7 DSP blocks for its FPGA implementation. The conventional Barrett algorithm uses 32 × 64 and 32 × 32 multipliers, which require 10 DSP blocks for its implementation for the same parameter set. The proposed word-level Montgomery algorithm, on the other hand, uses three 11 × 21 multipliers, which require 3 DSP blocks for its implementation for the same parameter set. Therefore, the proposed word-level Montgomery algorithm uses 57% and 70% less DSP blocks than the conventional Montgomery and Barrett algorithms, respectively, for the parameter set (1024,32). This shows the superiority of our approach.

### 3.2. The NTT unit

The proposed design uses NTT units, which implement the Gentleman-Sande butterfly configuration shown in steps 7–8 of the Algorithm 2 and steps 9–10 of the Algorithm 3. The proposed NTT unit is shown in Fig. 3 and it consists of one modular adder, one modular subtractor and one modular multiplier hardware. The first output, *Out0*, comes from the modular adder while the second output, *Out1*, comes from modular subtractor and multiplier. Due to 6 clock cycles latency of the modular multiplier, there is 6 clock cycles difference between two output coefficients. In order to synchronize both output coefficient, extra 6 flip-flops are placed at the output of modular adder hardware. The proposed NTT unit has 7 clock cycles latency and it is pipelined. It takes 3 coefficients as inputs and produces 2 coefficients as outputs per clock cycle after filling the pipeline.

The proposed NTT unit can also be configured to perform single modular multiplication operation by providing 0, first multiplicand and second multiplicand to the inputs *In0, In1* and *MultIn*, respectively. This configuration is used for performing coefficient-wise multiplication of polynomials in NTT domain and multiplying the coefficients of the polynomial with the powers of $\Psi$, $\Psi^{-1}$ and $n^{-1}$ in $\mathbf{Z}_q$. This configurability and re-use of NTT unit eliminate the need for extra modular multiplier for performing these operations. The proposed NTT unit can also be configured to perform modular addition and subtraction operations by reading *AddOut* and *SubOut* outputs.

### 3.3. The overall design

Determining the degree of parallelization in an architecture is not an easy task and it depends on the area and throughput requirements of the application. The NTT and INTT schemes shown in Algorithm 2 and Algorithm 3, respectively, allow the parallelization of the NTT and INTT operations by performing multiple Gentleman-Sande butterfly operations in parallel in one stage. An $n$-pt NTT consists of $\log_2(n)$ stages, where each stage has $(n/2)$ butterfly operations. Therefore, an NTT operation can be parallelized by performing multiple butterfly operations in one stage in parallel. In this work, hardware block performing one butterfly operation is referred as processing unit (PU).

The proposed architecture aims high performance with reasonable resources usage. Since the main building block of a polynomial

**Table 1**
Supported $K$ range for each parameter set.

| $(n, K)$ | Range |
|---|---|
| (256,16) | $9 < K \leq 18$ |
| (512,16) | $10 < K \leq 20$ |
| (1024,16) | $11 < K \leq 22$ |
| (1024,32) | $22 < K \leq 33$ |
| (2048,32) | $24 < K \leq 36$ |
| (4096,32) | $26 < K \leq 39$ |

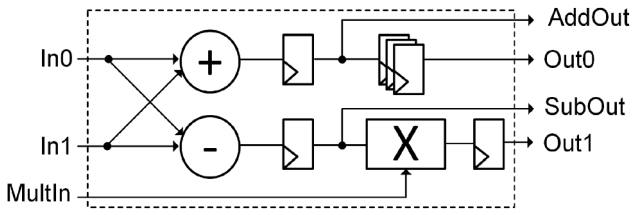



**Fig. 3.** NTT Unit.

multiplication operation is NTT, we analyze the performance of NTT operation with different number of PUs in order to decide the optimal number of PUs for a balanced design in terms of both area and throughput. We model the number of latency in terms of clock cycle as shown in Eq. (1).

$$\log_2(n) \times \left(\frac{n}{2 \times \text{PU number}} + 6\right) \tag{1}$$

Then, we plot the latency vs. $\log_2(n)$ graph for PU numbers ranging from 4 to 128 for the $n$ values in our parameter set as shown in Fig. 4a. We also plot the area $\times$ time vs. $\log_2(n)$ graph for PU numbers ranging from 4 to 128 for the $n$ values used in our parameter set as shown in Fig. 4b, where PU number and the latency in terms of clock cycles are used for area and time parameters, respectively. When large $n$ is used as required in homomorphic applications, the latency of NTT operation increases significantly for designs with 4, 8 and 16 PUs. Also, when $n$ is large, the designs with 4, 8 and 16 PUs show similar area $\times$ time performance with other designs as shown in Fig. 4b. When small $n$ is used as required in most lattice-based post-quantum cryptosystems, the designs with 64 and 128 PUs show worse area $\times$ time performance than other designs with similar latency performance as shown in Fig. 4b. As we aim a balanced architecture in terms of area and performance, we select the PU number as 32 in our design.

The overall architecture of the proposed run-time configurable NTT-based polynomial multiplier is shown in Fig. 5. The proposed architecture uses 32 PUs, where a PU performs steps 5–10 and steps 7–12 in Algorithm 2 and Algorithm 3, respectively. A PU consists of one NTT unit, 4 block RAMs (BRAMs) for storing the coefficients of polynomials (*POL0* and *POL1* in Fig. 5), 1 BRAM for storing the powers of $\omega$ and $\omega^{-1}$ to be used for NTT/INTT operations (*TW* in Fig. 5) and 1 BRAM for storing the powers of $\Psi$ and $\Psi^{-1}$ to be used for pre-processing and post-processing operations (*PSI* in Fig. 5). The proposed architecture also has an address/selection signal generator unit, which produces all necessary control signals for the NTT, INTT and polynomial multiplication operations.

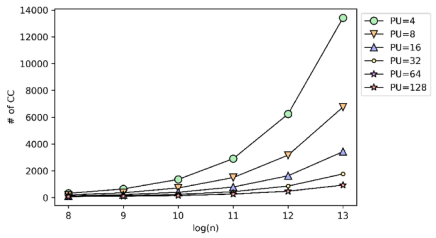
The proposed hardware architecture can be configured to perform 3 different operations: NTT, INTT and NTT-based polynomial multiplication. The NTT and INTT operations are performed as described in Algorithm 2 and Algorithm 3, respectively. The polynomial multiplication, on the other hand, can be performed in two different ways: *i*) both input polynomials are in the polynomial domain and *ii*) one of the input polynomials is in the polynomial domain while the other input polynomial is already in the NTT domain. The first and second polynomial multiplication methods will be referred as full polynomial multiplication (FPM) and half polynomial multiplication (HPM) for the rest of the paper. The proposed architecture can perform both FPM and HPM operations while our earlier work [8] can perform only HPM operation. Besides, the proposed architecture supports 6 different parameter sets $(n, K) = \{(256, 16), (512, 16), (1024, 16), (1024, 32), (2048, 32), (4096, 32)\}$, which are utilized in the lattice-based cryptosystems.

Managing complex memory access schedule is one of the most challenging parts of NTT-based polynomial multiplier architecture design. When a configurable architecture is aimed, this problem becomes more challenging since a flexible memory access schedule is required. Polynomials with different degrees ($n$) require different data alignment in memory and data access pattern. Therefore, the proposed memory access scheme in this work generates necessary control signals for storing and accessing polynomial coefficients as required for each parameter set. However, in our earlier work [8], the memory access scheme generates the same memory control signals for each polynomial input.

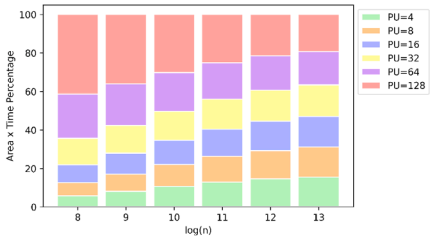
An $n$-pt NTT operation can be implemented using two $(n/2)$-pt NTT operations after the first stage of the $n$-pt NTT operation [9]. In this work, we exploit this property to design a configurable polynomial multiplier architecture in terms of $n$. Therefore, in this work which supports 256-pt to 4096-pt NTT/INTT, large NTT operations are performed using smaller NTT operations. For example, the proposed architecture with 32 PUs can perform one stage of 64-pt NTT with 7 clock cycles latency, where each PU performs one butterfly operation. Therefore, one 64-pt NTT with 6 stages will have a latency of $6 \cdot 7 = 42$ clock cycles. One 128-pt NTT operation then will perform its first stage in $(64/32) + 7 = 9$ clock cycles and two 64-pt NTT operations will be performed in $6 \cdot (2 + 7) = 54$ clock cycles as pipelined. In total, 128-pt NTT will be performed in $9 + 54 = 63$ clock cycles. Similarly, 256-pt NTT operation will be performed in $8 \cdot (4 + 7) = 88$ clock cycles. It should be noted that these calculations ignore delays and stalls due to control of the NTT operation in the implementation. INTT operation is also performed similarly.

Full polynomial multiplication operation uses two NTT, one INTT and four coefficient-wise multiplication of polynomials as shown in Algorithm 1. Similarly, half polynomial multiplication operation uses one NTT, one INTT and three coefficient-wise multiplication of polynomials. The number of clock cycles required to perform each operation by the proposed architecture for supported parameter sets are shown in Table 2.

The powers of $\omega$ and $\omega^{-1}$ are stored in 32 BRAMs as shown in Fig. 6. Since the proposed architecture divides an NTT operations into smaller NTT operations, only the twiddle factors necessary for performing the first stage of NTT operation for sizes from 2 to 4096 are stored. In total, $32 \cdot (64 + 32 + 16 + 8 + 4 + 2 + 1 + 1 + 1 + 1 + 1 + 1) = 4224$ twiddle factors are stored in 32 BRAMs. Besides, for INTT operation, 4224 inverse twiddle factors are stored in the other half of the same 32 BRAMs.




(a) Number of clock cycles (CC) vs. $\log_2(n)$ for Different PU Numbers




(b) Area×Time Percentage vs. $\log_2(n)$ for Different PU Numbers

**Fig. 4.** Number of clock cycles (CC) and Area $\times$ Time percentage estimations for different $n$ and PU numbers.

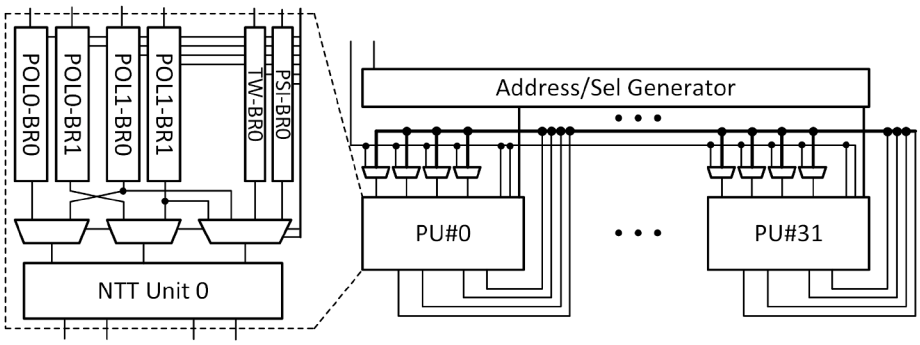



**Fig. 5.** The overall design.

Similarly, the powers of $\Psi/\Psi^{-1}$ values are stored in 32 BRAMs. Since the proposed work uses Montgomery modular reduction algorithm, the powers of $\omega$, $\omega^{-1}$, $\Psi$, $\Psi^{-1}$ and $n^{-1} \pmod q$ are multiplied with the necessary powers of Montgomery constant, $R$, prior to the FPGA in order to eliminate extra modular multiplications in run-time. The powers of $\omega$, $\omega^{-1}$, $\Psi$, $\Psi^{-1}$ and $n^{-1} \pmod q$ are loaded into the FPGA prior to any operation for a parameter set. If the parameter set is changed, new $\omega$, $\omega^{-1}$, $\Psi$, $\Psi^{-1}$ and $n^{-1} \pmod q$ values should be loaded.

For NTT/INTT operations, *POL1* and *PSI* BRAMs are not used while the full and half polynomial multiplication operations use all BRAMs. Polynomial multiplication operation requires the coefficients of resulting polynomial from INTT operation to be multiplied with the powers of $\Psi^{-1}$ for post-processing operation as shown in step 7 of Algorithm 1. INTT operation requires the coefficients of output polynomial to be multiplied with $n^{-1} \pmod q$ as shown in steps 19–21 in Algorithm 3. Therefore, we merged these two operations by multiplying the powers of $\Psi^{-1}$ with $n^{-1} \pmod q$ prior to loading precomputed coefficients into FPGA.

Since the proposed architecture uses 32 PUs and one PU takes two polynomial coefficients as inputs, 64 BRAMs are used to store one polynomial in the proposed architecture. The memory access patterns of the first two stages of 1024-pt NTT operation are shown in Fig. 7 as an example. In Fig. 7, the numbers in the BRAMs represent the indices of the polynomial coefficients. The NTT operation in Algorithm 2 starts butterfly operation with 0th and $(n/2)$th coefficients and continues with 1st and $((n/2)+1)$th coefficients. Since the coefficient pairs (0, 512) to (31, 543) need to be read in the same clock cycles, they are stored in different BRAMs in the proposed architecture. Due to the read/write pattern of the NTT algorithm, the coefficients read in the same stage should be stored in the same BRAMs for the next stage. For example, the coefficient pair (0, 512) read from *BR0* and *BR1* in the first stage should be stored in the *BR0* for the next stage as shown in Fig. 7. Since only one coefficient can be stored into one BRAM in a clock cycle, an extra register is placed at the output of modular multiplier hardware in the NTT unit as shown in Fig. 3. Therefore, both 0th and 512th coefficients can be stored into the same BRAM in two clock cycles. Since the proposed NTT unit is pipelined, this extra register does not affect the throughput of the proposed architecture.

Similarly, the coefficient pairs (32, 544) to (63, 575) need to be read and stored similar to the coefficient pairs (0, 512) to (31, 543) as shown in Fig. 7. This access pattern requires the swap of some coefficients in different memory blocks as shown with green boxes in Fig. 7. Therefore, we used an alternating memory access scheme to avoid collisions during the memory store operations.

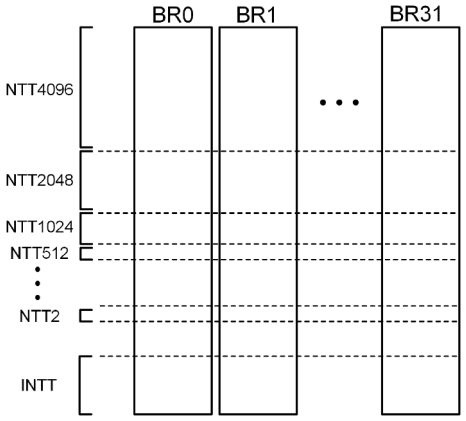



**Fig. 6.** BRAMs storing twiddle factors.

In this scheme, for an $n$-pt NTT operation, coefficients stored in the first and second halves of the memory blocks should be read in an alternating way. For example, in the first read operation of the first stage of NTT, the coefficients at address 0 should be read. Then, in the second read operation, coefficients at address $(n/128)$ should be read instead of address 1. Then, coefficients at addresses 1 and $(n/128)+1$ should be read consecutively and so on. Finally, coefficients at addresses $(n/128)-1$ and $(n/64)-1$ should be read consecutively to finish the first stage of NTT operation. For the next stage, the same memory pattern should be used for two $(n/2)$-pt NTT operations separately. All NTT operations with different sizes use the same alternating memory access scheme. The proposed flexible memory access scheme handles memory access operations for different $n$ values by generating necessary memory read/write signals.

NTT operation takes input polynomial in standard order and produces output polynomial in bit-reversed order while INTT takes input polynomial in bit-reversed order and produces output polynomial in standard order. Therefore, INTT operation uses the same memory access

**Table 2**
Number of clock cycles required for each operation and parameter set.

| | (*n*, *K*) | | | | | |
|---|---|---|---|---|---|---|
| | (256,16) | (512,16) | (1024,16) | (1024,32) | (2048,32) | (4096,32) |
| NTT | 104 | 153 | 250 | 250 | 451 | 876 |
| INTT | 121 | 178 | 291 | 291 | 524 | 1013 |
| HPM | 259 | 381 | 623 | 623 | 1121 | 2163 |
| FPM | 299 | 469 | 815 | 815 | 1537 | 3059 |

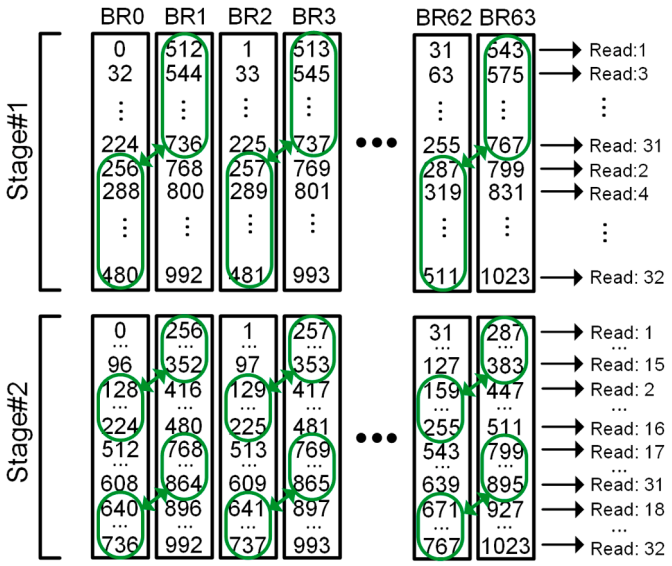



**Fig. 7.** Memory access pattern for 1024-pt NTT operation.

pattern in reverse order.

## 4. Hardware-software co-design framework

In order to show the use of the proposed polynomial multiplication architecture as an accelerator in lattice-based homomorphic applications utilizing polynomial multiplication operation, we designed a hardware/software co-design framework. For communication between the CPU and the FPGA, we utilized RIFFA driver [1], which employs a PCIe connection between CPU and FPGA.

As a case study, the proposed design is used to accelerate the decryption operation of the BFV scheme implemented in the SEAL homomorphic encryption library in a proof of concept accelerator framework. Although we utilized our proposed polynomial multiplier in decryption operation of the BFV scheme, it can be utilized for accelerating other homomorphic operations of the BFV scheme, which use polynomial multiplication. For example, encryption operation in the BFV scheme also uses polynomial multiplication as one of its core operations. However, it also requires sampling random polynomials and this necessitates the design and implementation of sampler hardware, which is out of scope of this work. In addition to that, the design in our earlier work [8] is shown to be effective as an accelerator for encryption operation in the BFV scheme by accelerating the polynomial multiplication operation in encryption operation for a fixed setting. Thus, we conclude that our proposed configurable polynomial multiplier can be used to accelerate other homomorphic operations including encryption.

The accelerator framework includes the SEAL software and an FPGA accelerator that implements our proposed polynomial multiplier architecture. The resulting framework is shown in Fig. 8.

### 4.1. A case study: SEAL by microsoft research

SEAL is an easy-to-use homomorphic encryption library developed by Cryptography Research Group at the Microsoft Research. SEAL implements two different homomorphic encryption schemes, BFV and the Cheon-Kim-Kim-Song (CKKS), for implementing homomorphic operations. Since we target demonstrating the acceleration of decryption operation in the BFV scheme of the SEAL, CKSS scheme is not be detailed in this section.

The BFV scheme [7] is an encryption scheme based on Ring Learning with Errors (RLWE) problem [14]. The RLWE problem is simply a ring based version of the LWE problem [15] and leads to the following encryption scheme as described in [14]. 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 $\Delta$ be $\lfloor q/t \rfloor$. Let $\mathbf{a} \xleftarrow{\$} \mathbf{S}$ and $\mathbf{a} \leftarrow \chi$ represent that $\mathbf{a}$ is uniformly sampled from the set $\mathbf{S}$ and sampled from discrete Gaussian distribution $\chi$, respectively. Secret key generation, public key generation, encryption and decryption operations described in textbook-BFV [7] 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:** $m \in R_t$,

$$m = \left[ \left\lfloor \frac{t}{q} [c_0 + c_1 \cdot s]_q \right\rceil \right]_t.$$

The decryption operation of the BFV scheme in the SEAL, shown in Algorithm 5, is implemented slightly different from the textbook-BFV, which requires division and rounding operations. In order to avoid these costly operations, SEAL uses full RNS variant of textbook-BFV for decryption operation [16], which requires base conversion as shown in step 4 of Algorithm 5. Decryption operation in the SEAL uses ciphertexts, secret key and a redundant modulus $\gamma \in \mathbb{Z}$. In the SEAL, secret key, $\bar{s}$, is stored in NTT domain and ciphertext polynomials, $c_0$ and $c_1$, used in decryption operation are stored in polynomial domain. The encoded plaintext output, $m$, is also generated and stored in polynomial domain.

Timing breakdowns of the decryption implementation in the SEAL for the parameter sets (1024,14), (1024,27) and (2048,29) are shown in Table 3. The average times for one NTT-based polynomial multiplication operation in the decryption operation of the SEAL are 26.2 μs, 28.8 μs and 54.6 μs for parameter sets (1024,14), (1024,27) and (2048,29), respectively. The timing results are average of 1000 executions. The timing results are obtained on an Intel i9-7900X CPU @ 3.30 GHz × 20 with 32 GB RAM using GCC version 7.5.0 in Ubuntu 16.04.6 LTS. As shown in Table 3, NTT-based polynomial multiplication operation forms 41.4% to 45.1% of execution time of one decryption operation.

In the SEAL library, there is a function called `decrypt`, which work as described in Algorithm 5. The `decrypt` function takes ciphertexts, $c_0$ and $c_1$, and secret key, $\bar{s}$, which is already in NTT domain, as inputs and outputs plaintext, $m$. In order to demonstrate the acceleration performance of the proposed polynomial multiplier architecture, we aim accelerating the decryption function of the SEAL for parameter sets (1024,14), (1024,27) and (2048,29) by offloading polynomial multiplication operation, $s.c_1$, into the FPGA that implements the proposed configurable NTT-based polynomial multiplier architecture. Since secret

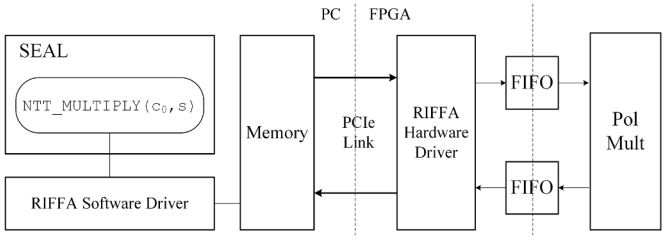



**Fig. 8.** Hardware/Software co-design framework.

key, $\bar{s}$, is already in NTT domain, only the polynomial $c_1$ is transformed into NTT domain using NTT operation. Then, $\overline{c_1}$ is coefficient-wise multiplied with the secret key and INTT operation is applied to transform the resulting polynomial, $\overline{s \cdot c_1}$, from NTT domain to polynomial domain. Therefore, this operation is a HPM.

We modified `decrypt` function of the SEAL to integrate our polynomial multiplier hardware as an accelerator into the SEAL. In the modified version, the `decrypt` function sends input polynomial $c_1$ to the FPGA, FPGA performs the polynomial multiplication $s \cdot c_1$ and returns the resulting polynomial to the CPU. The rest of the operations in `decrypt` function are performed in the CPU. Precomputed constants such as secret key, $\bar{s}$, and the powers of $\omega$, $\omega^{-1}$, $\Psi$, $\Psi^{-1}$ are sent to FPGA only once prior to any invocation of `decrypt` function.

To realize our framework, we use Xilinx VC707 Evaluation Kit, which includes a PCIe x8 Gen 2 Connector, with XC7VX485T-2FFG1761 FPGA. Xilinx IP Core 7-Series Integrated Block for PCIe provides a 128-bit interface with a 250 MHz clock, which has a 32 Gbps theoretical maximum bandwidth. As shown in Fig. 8, we utilize separate FIFO structures for data from the RIFFA driver and data to the RIFFA driver. This approach enables a pipelined architecture for maximizing the performance. 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 is directly sent to FPGA from memory. This way, cache memory is never trashed and running `decrypt` function does not affect the performance of other operations running on CPU.

SEAL library uses 64-bit integer type for storing the coefficients regardless of the actual bit-size of modulus, $q$. Since we can work with 16-bit or 32-bit coefficients, we can theoretically pack and send $128/16 = 8$ or $128/32 = 4$ coefficients per cycle, respectively. In order not to complicate memory access in CPU part, we pack and send $128/32 = 4$ coefficients per cycle instead. In [1], 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 Gbps. For the selected parameter sets in the SEAL, the polynomial multiplication in the decryption operation takes 1024 or 2048 coefficients as inputs and CPU can send $(3{\cdot}10^9)/(4{\cdot}1024) = 732420$ or $(3{\cdot}10^9)/(4{\cdot}2048) = 366210$ polynomial multiplication inputs per second, respectively, with 24 Gbps bandwidth.

**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: $c_1 s = \text{NTT\_MULTIPLY}\ (c_1, \overline{s})$
2: $c_t = (c_1 s + c_0) \cdot [\gamma \cdot t]_q$
3: **for** $m \in \{t, \gamma\}$ **do**
4: $\quad \mathbf{s}^{(m)} \leftarrow \text{FASTBCONV}(c_t, q, \{t, \gamma\}) \cdot [-q^{-1}]_m \ mod\ m$
5: **end for**
6: **for** $i$ from 0 by 1 to $n-1$ **do**
7: $\quad$ **if** $(\mathbf{s}^{(\gamma)}[i] > (\gamma/2))$ **then**
8: $\quad\quad m[i] = [\mathbf{s}^{(\gamma)}[i] - \mathbf{s}^{(\gamma)}[i] + \gamma]_t$
9: $\quad$ **else**
10: $\quad\quad m[i] = [\mathbf{s}^{(\gamma)}[i] - \mathbf{s}^{(\gamma)}[i]]_t$
11: $\quad$ **end if**
12: **end for**
13: **return** $m \leftarrow [m \cdot [\gamma^{-1}]_t]_t$
14: **function** NTT_MULTIPLY$(c_1, \overline{s})$
15: $\quad \overline{c_1} = \text{NTT}(c_1)$
16: $\quad c_1 s = \text{INTT}(\overline{c_1} \odot \overline{s})$
17: $\quad$ **return** $c_1 s$
18: **end function**
19: **function** FASTBCONV$(x, q, \beta)$
20: $\quad$ **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**

**Algorithm 5.** Decryption implementation in SEAL [2].

**Table 3**
Timing of decryption implementation in the SEAL.

| Operation | Time ($\mu$s) | Percentage (%) |
|---|---|---|
| | $n$=1024, $K$=14, $t$=8-bit, 256-bit security | |
| NTT_MULTIPLY | 26.2 | 41.4% |
| FASTBCONV | 17.7 | 27.9% |
| Others | 19.43 | 30.7 % |
| | $n$=1024, $K$=27, $t$=8-bit, 128-bit security | |
| NTT_MULTIPLY | 28.8 | 43.2% |
| FASTBCONV | 19.5 | 29.2% |
| Others | 17.4 | 27.6 % |
| | $n$=2048, $K$=29, $t$=8-bit, 256-bit security | |
| NTT_MULTIPLY | 54.6 | 45.1% |
| FASTBCONV | 34.5 | 28.5% |
| Others | 31.8 | 26.4 % |

Although the proposed work uses RIFFA driver utilizing PCIe connection for establishing communication between the host CPU and FPGA, programmable multiprocessor system on a chip (MPSoC) platforms with less communication cost can also be used based on the requirements of the target application.

## 5. Results

We developed the architecture described in this work into Verilog modules and realized it using Xilinx Vivado 2018.1 tool for the Xilinx VC707 Evaluation Kit, which has a Virtex-7 FPGA (XC7VX485T-2FFG1761). The core part of our proposed work uses 39.6K LUTs (9.2%), 21.1K DFFs (2.5%), 96 BRAMs (6.5%) and 224 DSPs (6.2%).

There are many works reported in the literature about the efficient implementation of NTT-based polynomial multiplication operation [8,12,17–26]. In [17] and [20], the authors propose an accelerator framework for homomorphic operations. They optimize their architecture for single $q$ and $n$. They also utilize a memory-based algorithm for integer modular reduction operation. Our proposed polynomial multiplier, on the other hand, supports multiple parameter sets and can be utilized as accelerator in different applications. The works in [8] and [26] present similar architectures with our proposed polynomial multiplier architecture. However, they support a single parameter set, which has a very low multiplicative depth for homomorphic encryption applications. The architecture in [18] works with very large $K$, namely $K$=512. However, compared to our work, it lacks parallelism due to large integer arithmetic. In [25], the authors propose an architecture for fixed $q$ and $n$, which enable a highly-optimized integer modular reduction architecture. However, their parameter set has low multiplicative depth and their architecture is not reconfigurable. The work in [19] utilizes the Karatsuba algorithm instead of NTT for polynomial multiplication. It works with single parameter set, which has a multiplicative depth of 4. Öztürk et al. proposes a highly parallelized NTT-based polynomial multiplier architecture with single parameter set [22]. They utilize the conventional Barrett algorithm for integer modular reduction and perform separate reduction operation for polynomial reduction. In [24], the authors implement a memory efficient NTT algorithm, which computes twiddle factors on-the-fly. However, it uses single processing unit and supports single parameter set. In [23], the authors implement the same polynomial multiplier architecture for four different parameter sets, where $n$ is ranging from 256 to 2048. However, they use single processing unit and our proposed polynomial multiplier architecture shows better performance than their work. In [21], the authors adopt a hardware generator tool for generating NTT hardware for a given parameter set. However, their generated NTT hardware does not support run-time configurability. Although some of these works perform slightly different operations than polynomial multiplication, we

only report the implementation results for the NTT and polynomial multiplication parts of these works. The implementation results of the works in the literature and the work proposed in this paper are reported in Tables 4 and 5, respectively.

The proposed configurable architecture in this paper is the only work in the literature supporting multiple $K$ and $n$ for FPGA platforms. Other works in the literature either are designed for fixed $q$ or do not support multiple $n$ values. The proposed work shows better area and timing performance than the most of the works in the literature. Although the works in [8,21,26] show better timing performance than the proposed work in this paper, they do not support multiple $q$ and $n$ values and the proposed work in this paper uses less FPGA resources. The polynomial multiplier in [12] shows both better area and timing performance; however, it is designed and optimized for fixed $q$. Although there are other implementations [27–29] supporting multiple $n$ and $q$ values, these works target ASIC platforms. Therefore, they are not included in the comparison.

Our proposed architecture is deployed into a framework that aims accelerating the decryption operation of the BFV scheme in the SEAL. For proof of concept, we select three parameter sets (1024,14), (1024,27), (2048,29) of the SEAL and offload polynomial multiplication operation in the `decrypt` function of the SEAL into the FPGA that implements our configurable NTT-based polynomial multiplier architecture. Then, we obtained performance numbers on a real CPU-FPGA heterogeneous application setting. The polynomial multiplication operation in the `decrypt` function of the SEAL with parameter sets (1024,14), (1024,27), (2048,29) yields 26.2 µs, 28.8 µs, 54.6 µs, respectively, in pure software implementation with host computer as specified in Section 4.1. Compared to the software, the proposed configurable polynomial multiplier architecture performs the same polynomial multiplication operations in 4.15 µs, 4.15 µs, 7.5 µs, respectively, excluding I/O operations. Compared to the software, we achieved up to 7 × speedup, excluding I/O operations for polynomial multiplication operation. With our setting, sending or receiving one polynomial of degree 1024 and 2048 from the CPU to FPGA via DMA takes 1.3 µs and 2.67 µs, respectively, on average. Therefore, our accelerator-based implementation, including I/O overhead, yields 6.75 µs, 6.75 µs, 12.96 µs, respectively, for selected parameter sets. Compared to the software, we achieved up to 4.2 × speedup, including I/O overhead. The `decrypt` function of the SEAL with parameter sets (1024,14), (1024,27), (2048,29) yields executions times of 63.33 µs, 65.7 µs, 120.9 µs, respectively, in pure software implementation as shown in Table 3. Compared to the software, the proposed framework with configurable polynomial multiplier architecture performs the same decryption operations in 43.93 µs, 43.7 µs, 79.26 µs, respectively, including I/O overhead. Compared to the software, we still achieve up to 1.52 × speedup, including I/O overhead for decryption operation.

The proposed polynomial multiplier architecture is shown to be effective as an accelerator for homomorphic applications. Since our proposed work aims supporting a range of applications with a focus on accelerating homomorphic operations, it is not meaningful to compare our architecture with highly-optimized hardware and/or software implementations of a specific NTT-friendly lattice-based post-quantum cryptosystems such as `CRYSTALS-Kyber (v1)` ([30,31]), `NewHope` ([5,31]) and `qTESLA` ([4]).

The proposed NTT-based polynomial multiplier architecture supports lattice-based post-quantum cryptosystems with NTT-friendly parameter set [3–5]. However, not all lattice-based post-quantum cryptosystems such as SABER [32] supports NTT operation due to their parameter sets. The proposed architecture in this paper can be extended to perform polynomial multiplication for cryptosystems without NTT-friendly coefficient modulus with slight modifications as shown in [11]. A unified polynomial multiplier architecture can be the focus of a future work.

Although the proposed work supports relatively small parameter sets for homomorphic operations, homomorphic multiplication in particular, and has a low multiplicative depth, our polynomial multiplier can be easily utilized in designs with larger degrees and depths. Many works [2,17,20] use one large modulus ($Q$), which is mapped into smaller coprime moduli ($q_0$, $q_1$, $q_2$…). In order to increase the parallelism, they employ Chinese Remainder Theorem (CRT) [33], which transforms each coefficient in $Q$ into multiple smaller coefficients in $\{q_0, q_1, q_2\ldots\}$, and perform operations in small moduli separately in parallel instead of in $Q$. For example, the work in [20] with multiplicative depth of 4 uses six small 30-bit coprime moduli instead of one large 180-bit modulus with

**Table 4**
Comparative table (FPGA Resources).

| Work | Platform | $n$ | $K$ | LUT/DSP/BRAM | Clock (MHz) |
|---|---|---|---|---|---|
| [17] | Virtex-6 | 65,536 | 30 | 72K / 250 / 106 | 100 |
| [18] | Virtex-7 | 4096 | 125 | 69K / 144 / – | 100 |
| [19] | Stratix-V | 2560 | 125 | 30K / 100 / – | 331 |
| [20] | Zynq | 4096 | 30 | 64K / 200 / 400 | 225 |
| [22] | Virtex-7 | 32,768 | 32 | 219K / 768 / 193 | 250 |
| [25][a] | Spartan-6 | 1024 | 30 | 1644 / 1 / 6.5 | 200 |
| [24][a] | Spartan-6 | 1024 | 17 | 250 / 3 / 2 | – |
| | | | | 240 / 3 / 2 | |
| | | | | 250 / 3 / 2 | |
| [23][a] | Spartan-6 | 256 | 21 | 2829 / 4 / 4 | 247 |
| | | 1024 | 31 | 6689 / 4 / 8 | 241 |
| [21] | Virtex-7 | 4096 | 30 | 54K / 517 / 208 | 200 |
| [12][a] | Spartan-6 | 256 | 21 | 14K / 128 / 1 | 233 |
| | | 512 | 23 | 18K / 128 / 2.5 | 200 |
| [26] | Virtex-7 | 1024 | 32 | 77K / 952 / 325.5 | 200 |
| | | | | 67K / 599 / 129 | |
| [8] | Spartan-6 | 1024 | 32 | 1.2K / 14 / 14 | 212 |
| | Virtex-7 | | | 33.8K / 476 / 227.5 | 200 |
| **TW**[b] | Virtex-7 | 256 | 16 | 39.6K / 224 / 96 | 150 |
| | | 512 | | | |
| | | 1024 | | | |
| | | 1024 | 32 | | |
| | | 2048 | | | |
| | | 4096 | | | |

[a] Uses fixed $q$;
[b] This Work (excluding RIFFA Hardware Driver and input/output FIFOs shown in Fig. 8).

**Table 5**
Comparative table (Performance).

| Work | Platform | $n$ | $K$ | Latency (µs) | |
|---|---|---|---|---|---|
| | | | | NTT | Pol.Mul. |
| [17] | Virtex-6 | 65,536 | 30 | – | 3376 |
| [18] | Virtex-7 | 4096 | 125 | – | 1960 |
| [19] | Stratix-V | 2560 | 125 | – | 583 |
| [20] | UltraScale | 4096 | 30 | 73 | 171 |
| [22] | Virtex-7 | 32,768 | 32 | 51 | 152 |
| [25][a] | Spartan-6 | 1024 | 30 | – | 110 |
| [24][a] | Spartan-6 | 1024 | 17 | – | 25 |
| | | | | | 50 |
| | | | | | 100 |
| [23][a] | Spartan-6 | 256 | 21 | – | 6 |
| | | 1024 | 31 | – | 33 |
| [21] | Virtex-7 | 4096 | 30 | – | 10 |
| [12][a] | Spartan-6 | 256 | 21 | – | 0.94 |
| | | 512 | 23 | – | 1.77 |
| [26] | Virtex-7 | 1024 | 32 | 0.4 | 0.96 |
| | | | | 0.7 | 1.40 |
| [8] | Spartan-6 | 1024 | 32 | – | 37.67 |
| | Virtex-7 | | | – | 1.25 |
| **TW**[b] | Virtex-7 | 256 | 16 | 0.69 | 1.99 |
| | | 512 | | 1.02 | 3.12 |
| | | 1024 | | 1.66 | 5.43 |
| | | 1024 | 32 | 1.66 | 5.43 |
| | | 2048 | | 3.01 | 10.25 |
| | | 4096 | | 5.84 | 20.39 |

[a] Uses fixed $q$;
[b] This work.

$n$=4096. Similarly, SEAL uses the same approach for implementing homomorphic operations. Therefore, the polynomial multiplier proposed in this work with small parameter set can be utilized for accelerating operations with larger parameter sets.

Our run-time configurable architecture is optimized for the entire proof of concept framework. Therefore, the frequency that we are using does not need to be the maximum possible frequency. Our datapath speed is optimized according to the I/O bandwidth of the PCIe connection. Increasing the frequency any further will also make our datapath faster than I/O, which will still add stall cycles to our pipeline. The pipeline stages of the proposed architecture can be modified accordingly for the requirement of target platform.

## 6. Conclusion

In this paper, we present the design and FPGA implementation of a run-time configurable and highly parallelized NTT-based polynomial multiplier architecture, which is shown to be effective as an accelerator for lattice-based homomorphic schemes. The proposed architecture supports 6 different parameter sets and it can be used as an accelerator for diverse number of applications ranging from post-quantum cryptography to homomorphic encryption. To the best of our knowledge, our configurable NTT-based polynomial multiplier architecture is the only FPGA implementation supporting multiple $q$ and $n$ values.

For proof of concept, we utilize our architecture in an framework for the BFV homomorphic encryption scheme, adopting a hardware/software co-design approach. Polynomial multiplication operation in the decryption operation of the BFV scheme implemented in the SEAL are offloaded to the accelerator in the FPGA via PCIe bus while the rest of operations in the decryption operation of the BFV scheme are executed in software running on an off-the-shelf desktop computer. We realized the framework on an FPGA operating with the SEAL software and the proposed framework accelerates the decryption operation in the SEAL. We used Xilinx VC707 board utilizing a Virtex-7 FPGA for our implementation. We improved the latency of NTT-based polynomial multiplications in decryption operation by up to 7 $\times$ and 4.2 $\times$ compared to its pure software implementation, excluding and including I/O overhead, respectively. In addition to that, we improved the latency of decryption operation by up to 1.52 $\times$ compared to its pure software implementation, including I/O overhead.

Finally, although the proposed architecture is shown to be effective for accelerating decryption operation of the BFV scheme implemented in the SEAL homomorphic encryption library, it can easily be utilized for accelerating post-quantum key encapsulation/signature algorithms and other homomorphic operations such as homomorphic multiplication, which use excessively many polynomial multiplications. As future work, we target accelerating other homomorphic operations such as homomorphic multiplication and post-quantum key-exchange protocols by implementing most of the operations in the FPGA. This will also lead to higher speedup values for homomorphic operations as less communication time will be required between FPGA and CPU compared to total execution time.

## Declaration of Competing Interest

No conflict of interest among editors listed at https://www.journals.elsevier.com/microprocessors-and-microsystems/editorial-board.

## Acknowledgement

We thank anonymous reviewers for their thoughtful comments. Dr. Öztürk and Dr. Savaş are supported by TUBITAK under grant number 118E725.

## References

[1] M. Jacobsen, Y. Freund, R. Kastner, RIFFA: A reusable integration framework for FPGA accelerators. 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, 2012, pp. 216–219, https://doi.org/10.1109/FCCM.2012.44.

[2] Microsoft SEAL (release 3.2), 2019, (https://github.com/Microsoft/SEAL), Microsoft Research, Redmond, WA.

[3] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J.M. Schanck, P. Schwabe, G. Seiler, D. Stehlé, CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM, 2017, (Cryptology ePrint Archive, Report 2017/634), https://eprint.iacr.org/2017/634.

[4] E. Alkim, P.S.L.M. Barreto, N. Bindel, J. Kramer, P. Longa, J.E. Ricardini, The Lattice-Based Digital Signature Scheme qTESLA, 2019, (Cryptology ePrint Archive, Report 2019/085), https://eprint.iacr.org/2019/085.

[5] E. Alkim, P. Jakubeit, P. Schwabe, Newhope on arm cortex-m. International Conference on Security, Privacy, and Applied Cryptography Engineering, Springer, 2016, pp. 332–349.

[6] C. Gentry. A Fully Homomorphic Encryption Scheme, Stanford, CA, USA, AAI3382729, 2009. Ph.D. thesis.

[7] J. Fan, F. Vercauteren, Somewhat Practical Fully Homomorphic Encryption, 2012, (Cryptology ePrint Archive, Report 2012/144), https://eprint.iacr.org/2012/144.

[8] A.C. Mert, E. Ozturk, E. Savas, Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture. 2019 22nd Euromicro Conference on Digital System Design (DSD), 2019, pp. 253–260, https://doi.org/10.1109/DSD.2019.00045.

[9] E. Chu, A. George, Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms, CRC press, 1999.

[10] J.M. Pollard, The fast Fourier transform in a finite field, Math. Comput. 25 (114) (1971) 365–374.

[11] W. Dai, B. Sunar, cuHE: a homomorphic encryption accelerator library, in: E. Pasalic, L.R. Knudsen (Eds.), Cryptography and Information Security in the Balkans, Springer International Publishing, Cham, 2016, pp. 169–186.

[12] X. Feng, S. Li, S. Xu, RLWE-Oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm, IEEE Transactions on Circuits and Systems II: Express Briefs (2019), https://doi.org/10.1109/TCSII.2019.2917621.1–1

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

[14] V. Lyubashevsky, C. Peikert, O. Regev, On ideal lattices and learning with errors over rings, in: H. Gilbert (Ed.), Advances in Cryptology – EUROCRYPT 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 1–23.

[15] O. Regev, On lattices, learning with errors, random linear codes, and cryptography. Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, in: STOC '05, ACM, New York, NY, USA, 2005, pp. 84–93, https://doi.org/10.1145/1060590.1060603.

[16] J.-C. Bajard, J. Eynard, M.A. Hasan, V. Zucca, A full RNS variant of FV like somewhat homomorphic encryption schemes, in: R. Avanzi, H. Heys (Eds.), Selected Areas in Cryptography – SAC 2016, Springer International Publishing, Cham, 2017, pp. 423–442.

[17] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, I. Verbauwhede, HEPCloud: an FPGA-based multicore processor for FV somewhat homomorphic function evaluation, IEEE Trans. Comput. 67 (11) (2018) 1637–1650, https://doi.org/10.1109/TC.2018.2816640.

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

[19] V. Migliore, M.M. Real, V. Lapotre, A. Tisserand, C. Fontaine, G. Gogniat, Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm, IEEE Trans. Comput. 67 (3) (2018) 335–347, https://doi.org/10.1109/TC.2016.2645204.

[20] S.S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, I. Verbauwhede, FPGA-based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data, 2019, (Cryptology ePrint Archive, Report 2019/160).

[21] J. Cathebras, A. Carbon, P. Milder, R. Sirdey, N. Ventroux, Data flow oriented hardware design of RNS-based polynomial multiplication for SHE acceleration, IACR Trans. CHES 2018 (3) (2018) 69–88.

[22] E. Ozturk, Y. Doroz, E. Savas, B. Sunar, A custom accelerator for homomorphic encryption applications, IEEE Trans. Comput. 66 (1) (2017) 3–16, https://doi.org/10.1109/TC.2016.2574340.

[23] D.D. Chen, N. Mentens, F. Vercauteren, S.S. Roy, R.C.C. Cheung, D. Pao, I. Verbauwhede, High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems, IEEE Trans. Circuits Syst. I 62 (1) (2015) 157–166, https://doi.org/10.1109/TCSI.2014.2350431.

[24] A. Aysu, C. Patterson, P. Schaumont, Low-cost and area-efficient FPGA implementations of lattice-based cryptography. 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 2013, pp. 81–86, https://doi.org/10.1109/HST.2013.6581570.

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

[26] A.C. Mert, E. Ozturk, E. Savas, Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme, IEEE Trans. Very Large Scale Integr. VLSI Syst. (2019) 1–10, https://doi.org/10.1109/TVLSI.2019.2943127.

[27] U. Banerjee, A. Pathak, A.P. Chandrakasan, 2.3 An energy-efficient configurable lattice cryptography processor for the quantum-secure internet of things. 2019 IEEE ISSCC, 2019, pp. 46–48, https://doi.org/10.1109/ISSCC.2019.8662528.
[28] S. Song, W. Tang, T. Chen, Z. Zhang, LEIA: A 2.05mm2140mW lattice encryption instruction accelerator in 40nm CMOS. 2018 IEEE Custom Integrated Circuits Conference (CICC), 2018, pp. 1–4, https://doi.org/10.1109/CICC.2018.8357070.
[29] T. Fritzmann, J. Seplveda, Efficient and flexible low-power NTT for lattice-based cryptography. 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2019, pp. 141–150, https://doi.org/10.1109/HST.2019.8741027.
[30] L. Botros, M.J. Kannwischer, P. Schwabe, Memory-efficient high-speed implementation of Kyber on Cortex-M4. Int. Conference on Cryptology in Africa, Springer, 2019, pp. 209–228.
[31] G. Seiler, Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography, IACR Cryptol. ePrint Arch. 2018 (2018) 39.
[32] J.-P. D'Anvers, A. Karmakar, S.S. Roy, F. Vercauteren, Saber: module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. International Conference on Cryptology in Africa, Springer, 2018, pp. 282–305.
[33] D. Boneh, et al., Twenty years of attacks on the RSA cryptosystem, Not. AMS 46 (2) (1999) 203–213.

**Ahmet Can Mert** received the B.S. and M.S. degree in electronics engineering from Sabanci University in 2015 and 2017, respectively. Currently, he is working towards the Ph.D. degree in electronics engineering at Sabanci University. His research interest include cryptographic hardware design, computer arithmetic and embedded systems.

**Erdinç Öztürk** received the B.S. degree in microelectronics from Sabanci University in 2003. He received the M.S. degree in electrical engineering in 2005 and Ph.D. degree in electrical and computer engineering in 2009 from Worcester Polytechnic Institute. After receiving his Ph.D. degree, he was with Intel in Massachusetts for almost 5 years as a hardware engineer, before joining Istanbul Commerce University as an assistant professor. He joined Sabanci University in 2017 as an assistant professor. His research interest include cryptographic hardware design and he focused on efficient identity based encryption implementations.

**Erkay Savaş** received the B.S. and M.S. degrees in electrical engineering from the Electronics and Communications Engineering Department, Istanbul Technical University in 1990 and 1994, respectively. He received the Ph.D. degree from the Department of Electrical and Computer Engineering, Oregon State University in June 2000. He has been a faculty member at Sabanci University since 2002. His research interests include applied cryptography, data and communication security, security and privacy in data mining applications, embedded systems security, and distributed systems.