ReMCA: A Reconfigurable Multi-Core Architecture for Full RNS Variant of BFV Homomorphic Evaluation

Yang Su, Bai-Long Yang, Chen Yang, *Member, IEEE*, and Song-Yin Zhao

***Abstract*— Fully homomorphic encryption (FHE) allows arbitrary computation on encrypted data and thus has potential in privacy-preserving computing. However, efficiency is still the bottleneck. In this paper we present an area-efficient and highly unified reconfigurable multi-core architecture (named ReMCA) for full Residue Number System (RNS) variant of Fan-Vercauteren variant of Brakerski's scheme (RNS-BFV), which employs a variable number of reconfigurable processing elements (PEs) and RNS channels. The PE unit can be flexibly configured as NTT, INTT or modular multiplier, thereby avoiding the need of other extra computational units. To reduce the computational complexity, ReMCA merges the pre/post-processing into NTT/INTT and unifies the read/write structure of NTT and INTT. Also, a conflict-free memory access pattern that doesn't need separate bit-reversal operation is proposed to optimize the memory access. Furthermore, targeting different computational requirements, a unified hardware architecture mapping model and data memory organization model are introduced, and all the computing units that RNS-BFV involved are optimized and mapped on ReMCA. ReMCA is evaluated on a Xilinx Virtex-7 FPGA platform. Running at 250MHz, it can perform 2260 homomorphic multiplication per second. When normalized to the same parameter set, the throughput and Area-Time-Products (ATPs) of ReMCA achieve 1.45×∼5.51× and 1.58×∼5.12× improvements.**

***Index Terms*— Homomorphic encryption, reconfigurable PE, multi-core architecture, NTT/INTT, RNS, BFV scheme.**

I. Introduction

CLOUD computing has fundamentally changed the commercial modes of today, but the cloud service provider may abuse the business data of its clients for gaining greater benefits, which brings a great risk of data privacy disclosure. Fully homomorphic encryption (FHE), an encryption scheme which enable evaluating arbitrary functions on encrypted data without knowing the secret keys, is an ideal solution for this privacy issue. By adopting FHE, the cloud service provider guarantees that personal private data cannot be decrypted while providing the computing services. Therefore, FHE has become a promising privacy-preserving solution for numerous applications, such as personal information sorting and retrieval [1], health data management [2] and machine learning [3] *et al*.

The concept of FHE was introduced by Rivest, Adleman and Dertouzos [4] in 1978 while the first functioning FHE scheme was constructed by Gentry [5] in 2009. Despite the groundbreaking work, these first-generation schemes were extremely slow and suffered from a massive ciphertext expansion problem. Since then, many more efficient FHE schemes based on learning with error (LWE) [6] or the ring-LWE (RLWE) [7] with reduced computation overhead and ciphertext size are proposed. However, performance is still the main factor that restricting the practicability of FHE, hence, the case for acceleration is much stronger than conventional cryptosystems. Among these schemes, the Fan-Vercautern variant [7] of Brakerski's scale-invariant scheme [8], which we collectively call BFV scheme, is one of the most promising and efficient FHE scheme in terms of evaluating the exact arithmetic over integers, and its acceleration has become an important research area.

There is a wealth of software implementations of BFV scheme. [9] implemented the BFV encryption scheme in C++ code using the arithmetic library FLINT [10]. It performs a homomorphic multiplication of BFV in 148 ms for polynomial degree 4096 with 127-bit coefficients. An improved version of BFV scheme was proposed in [11] using FLINT and NFLlib library respectively for the same polynomial degree with 124-bit modulus. As a result, performing a homomorphic multiplication it takes 91.2 ms and 17.2 ms respectively. To avoid the operations on very large numbers, the residue number system (RNS)-based BFV (RNS-BFV) schemes are also proposed to split the polynomial coefficient into several smaller ones, which enable the BFV to have high potential parallelism. In particular, [12] reported an homomorphic multiplication of RNS-BFV (i.e., BEHZ) for 4096-degree polynomial with 180-bit in 15.61 ms. Then, an improved version of [12] (i.e., HPS) was presented in [13]. They claims that for the same polynomial degree with 110-bit, the homomorphic multiplication is performed in 10.1 ms. However, even so, the software implementations still suffer from slow processing speed on conventional software platforms, which render them impractical for many critical FHE applications.

Manuscript received December 6, 2021; revised March 6, 2022; accepted March 25, 2022. Date of publication April 11, 2022; date of current version June 29, 2022. This work was supported in part by the National Natural Science Foundation (NNSF) of China under Grant 62176206 and Grant 62172436, in part by the Aeronautical Science Foundation of China under Grant 2020Z066070001, and in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2019B010154002. This article was recommended by Associate Editor M. Mozaffari Kermani. *(Corresponding author: Chen Yang.)*

Yang Su is with the School of Operational Support, Rocket Force University of Engineering, Xi'an 710025, China, and also with the School of Cryptography Engineering, Engineering University of People's Armed Police, Xi'an 710086, China (e-mail: wj_suyang@126.com).

Bai-Long Yang is with the School of Operational Support, Rocket Force University of Engineering, Xi'an 710025, China (e-mail: wj_suyang@126.com; xa_403@163.com).

Chen Yang is with the School of Microelectronics, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: chyang00@xjtu.edu.cn).

Song-Yin Zhao is with the School of Cryptography Engineering, Engineering University of People's Armed Police, Xi'an 710086, China (e-mail: zhaosongyin@163.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2022.3163970.

Digital Object Identifier 10.1109/TCSI.2022.3163970

Several hardware architectures have been proposed to accelerate the BFV scheme using FPGAs. [14] presented a hardware implementation of a residue polynomial multiplier (RPM) via a negative wrapped convolution (NWC) to accelerate the RNS-BFV, in which the twiddle factors are generated by a specially designed generator. A modified iterative NTT and a four-step NTT hardware architecture was introduced in [15] to accelerate the encryption and decryption operations of BFV scheme in simple encrypted arithmetic library (SEAL). A FPGA-based multi-core hardware accelerator for RNS-BFV was presented in [16] to speed up the homomorphic evaluations using efficient parallel NTT cores. To satisfy the practical application, they further proposed domain specific multi-core architectures for homomorphic evaluation of RNS-BFV using Arm + FPGA heterogeneous platform [17] and Amazon AWS massive size FPGA [18], respectively. Although the performance of aforementioned works has been improved a lot and the core components are specially optimized, the implementations of the RNS-based relinearization are rarely presented. More importantly, the computing units in each RNS channel or between RNS channel always lack configurability even if they have many common units. This reduces the resource utilization of common units, and also reduces the flexibility and scalability of the hardware.

In BFV and RLWE-based FHE schemes, NTT is one of the most expensive operations. Several hardware implementations focus on accelerating NTT from performance, efficiency, and flexibility perspective. To accelerate the polynomial multiplication of RLWE and SHE cryptosystems, [19] presented a Constant-Geometry (CG) NTT multiplier with optimized parameter sets and dataflow. [20] and [21] introduced a parametric NTT hardware generator that takes arithmetic configurations and the number of processing element*s* (PEs) as inputs to produce an efficient hardware for the given parameters. By doubling the transform throughput and generating the twiddle factors on the fly, an optimized *ping-pong* NTT algorithm and architecture was proposed in [22] to accelerate the large integer multiplication in FHE. In order to accelerate the CKKS scheme, [23] proposed a multi-core NTT/INTT architecture with optimized conflict-free memory access pattern. The work in [14] and [24] presented a hardware implementation of RPM based on a RNS friendly NTT architecture, which locally computes the required twiddle factors. In addition, NTT/INTT hybrid architectures were introduced in [25] and [26] to accelerate the polynomial multiplications in the post-quantum cryptography. However, the NTT/INTTs mentioned above always require separate pre/post-processing steps or additional multiplicative factors, which bring extra computational overhead and increase memory occupation. Besides, the PEs or butterfly units always lack configurability to support various computing units of BFV scheme, and the area efficiency is also relatively low.

To the best of our knowledge, there appear to be very few pure hardware implementations of full RNS-BFV scheme that support the configuration of both the PE parallelism and the RNS channel parallelism simultaneously. In order to realize the variability of the number of PEs and RNS channels, as well as meet the diversified computational requirements of FHE applications, this paper presents an area-efficient and highly unified reconfigurable multi-core architecture (named ReMCA) for homomorphic evaluation of full RNS-BFV, and implements it on Xilinx Virtex-7 FPGA platform. The contributions of this paper are summarized as follows.

- A unified low-complexity CG NTT/INTT algorithm without pre/post-processing is presented. To reduce the computational complexity and memory occupation of twiddle factors, we merge the pre-processing and post-processing operations into the main NTT and INTT, respectively. Also, we unify the NTT and INTT in the same algorithm and settle the read/write structure inconsistency problems between them. As a result, we can not only eliminate the pre/post-processing steps in NTT/INTT, but also save the twiddle factors memory overhead and unify the memory access structure.
- An area-efficient and highly unified reconfigurable multi-core architecture named ReMCA is proposed for full RNS-BFV. In ReMCA, we present a unified dynamic reconfigurable PE unit which can be flexibly configured as a butterfly unit for the NTT/INTT or a modular multiplier for the modular operations in other computing units. The PE unit avoids the need of extra computational units for RNS-BFV homomorphic evaluation, which helps save the hardware resources and eventually improves the area efficiency. Besides, a conflict-free memory access pattern that doesn't need the separate bit-reversal operation is proposed to optimize the memory access.
- A unified computational model and high-performance mapping methods of different computing units that RNS-BFV homomoprhic evaluation involved are proposed. By optimizing the data flow diagram (DFG) and control flow diagram (CFG), we present a unified hardware architecture mapping model and data memory organization model based on ReMCA. On the basis of optimizing the computing units, we further present the high-performance mapping methods and overall execution flow for these units, thereby improving the overall performance of homomorphic evaluation.

The rest of the paper is organized as follows: Section II introduces the necessary preliminaries. Section III provides the optimized algorithms for NTT and basis extension/scaling. In Section IV, the reconfigurable multi-core architecture and unified computing model are proposed. Section V presents the algorithm mapping methods and overall execution flow. Section VI provides the implementation results and compares them with the counterparts. Section VII concludes the paper.

II. Preliminaries

A. The Textbook BFV Scheme

The main procedures of the textbook BFV scheme are enumerated as follows. For details of the functions, interested readers may follow the original paper [7].

BFV. KeyGen(λ, w): For a given security parameter λ, choose the polynomial degree, plaintext/ciphertext modulus, and Gaussian distribution χ_σ. Sample the polynomial $s \leftarrow R_2$, the coefficients of which are from $\{-1, 0, 1\}$, return the secret key $\mathbf{sk} = (1, s) \in R_2^2$. The public key $\mathbf{pk}$ is a pair of polynomials $(pk_0, pk_1) = (-[a \cdot sk + e]_q, a)$, where $a \leftarrow R_q$ and $e \leftarrow \chi_\sigma$. The relinearization keys $\mathbf{rlk}$ is a set of $(l+1)$ pairs of polynomials generated as follows: for $0 \le i \le l$, $\mathbf{rlk}[i] = ([-(a_i \cdot s + e_i)]_q + w^i s^2, a_i)$, where $a_i \leftarrow R_q$, $e_i \leftarrow \chi_\sigma$, $l = \lfloor \log_w q \rfloor$, and $w \in \mathbb{Z}$ is a decomposition base used to express a polynomial in R_q in terms of $(l+1)$ polynomials. The procedure outputs the tuple: $(\mathbf{sk}, \mathbf{pk}, \mathbf{rlk})$.

BFV. Enc$(m, \mathbf{pk})$: Take a plaintext polynomial $m \in R_t$, $\Delta = \lfloor q/t \rfloor$, and sample the error polynomials $u \in R_2$ and e_1, $e_2 \in \chi_\sigma$. Then compute the ciphertext $\mathbf{ct} = ([\mathbf{pk}[0] \cdot u + e_1 + \Delta m]_q, [\mathbf{pk}[1] \cdot u + e_2]_q)$.

BFV. Dec$(\mathbf{ct}, \mathbf{sk})$: Output the plaintext polynomial $m = \left[\frac{t}{q} \lfloor [\mathbf{ct}[0] + \mathbf{ct}[1] \cdot s]_q \rceil \right]_t$.

BFV. HomAdd$(\mathbf{ct}_0, \mathbf{ct}_1)$: The homomorphic addition is computed by $\mathbf{ct}_{add} = ([\mathbf{ct}_0[0] + \mathbf{ct}_1[0]]_q, [\mathbf{ct}_0[1] + \mathbf{ct}_1[1]]_q)$.

BFV. HomMult$(\mathbf{ct}_0, \mathbf{ct}_1, \mathbf{rlk})$: The homomorphic multiplication consists of two steps:

(1) The first step computes tensor products $\mathbf{c}_\tau$, with $\tau \in \{0, 1, 2\}$, such that:

$c_0 = [\lfloor (t/q) \cdot \mathbf{ct}_0[0] \cdot \mathbf{ct}_1[0] \rceil]_q$,

$c_1 = [\lfloor (t/q) \cdot (\mathbf{ct}_0[0] \cdot \mathbf{ct}_1[1] + \mathbf{ct}_0[1] \cdot \mathbf{ct}_1[0]) \rceil]_q$,

$c_2 = [\lfloor (t/q) \cdot \mathbf{ct}_0[1] \cdot \mathbf{ct}_1[1] \rceil]_q$.

(2) The second step decomposes c_2 in base w as $c_2 = \sum_{i=0}^{l} c_2^{(i)} w^i$ and returns $\mathbf{ct}_{mul}[j]$, with $j \in \{0, 1\}$, such that $\mathbf{ct}_{mul}[j] = \left[c_j + \sum_{i=0}^{l} \mathbf{rlk}[i][j] \cdot c_2^{(i)}\right]_q$.

B. Parameter Setup

A larger parameter set implies greater multiplicative depth. In order to support the relatively practical applications, we design ReMCA to support a multiplicative depth of 6 and at least 80-bit security using the RNS techniques of HPS [13]. To this end, we set the polynomial degree N to 4096, the size of modulus q to 128-bit, the standard deviation of the Gaussian distribution to $\sigma = 3.19$ and the size of the larger modulus Q to at least 288-bit. More concretely, we use 32-bit primes to construct the RNS for our implementation. Thus, the modulus q is taken as a product of four 32-bit primes, while Q is taken as a product of nine 32-bit primes. For RNS representation, the basis extension and scaling operations are required to switch from one RNS to another when the coefficients are moved from modulo q to modulo Q or vice versa.

III. Algorithm and Approach

In this section, we describe the main challenges that we faced when constructing ReMCA for full RNS-BFV scheme and the optimized algorithms that we took to address them.

A. Unified Low-Complexity NTT/INTT

The acceleration of polynomial multiplication directly determines the performance of homomorphic multiplication. Compared with the Schoolbook algorithm [27] that has a quadratic complexity and the Karatsuba algorithm has a sub-quadratic complexity [28], the NTT-based polynomial multiplication has the lowest complexity $O(N \log N)$. For a polynomial multiplication, two NTT transforms, one point-wise multiplication, and one INTT transform are required to perform a complete polynomial multiplication. Nevertheless, NTT itself remains a bottleneck.

One generally employs the NWC method to compute the product of two polynomials in R_q to avoid the zero-padding operations and doubling the length of polynomials. However, this will bring an extra pre-processing step that multiplying the input polynomials by N powers of pre-processing factors, as well as an extra post-processing step that multiplying the output polynomial by N powers of post-processing factors and N^{-1}, which greatly increases the computational complexity and the memory occupation of twiddle factor. Besides, compared to the Cookey-Tukey (CT) algorithm, the read/write control and non-conflict memory access pattern of Constant-Geometry (CG) algorithm are significantly simplified. Therefore, to reduce the computational complexity and provide better scalability for ReMCA, we present a novel unified low-complexity NTT/INTT algorithm for CG algorithm as Algorithm 1 shows.

When straightforwardly calculating the NTT as [29], it requires $(N/2) \log_2 N$ modular multiplications and a pre-processing step involving N modular multiplications. To eliminate the pre-processing step, we merge it into the main NTT by replacing the original twiddle factor term $\omega_N^{k_1}$ with $\psi_{2N}^{2k_1+k_2}$ (in step 7 and step 8). As a result, the CG NTT requires only $(N/2) \log_2 N$ modular multiplications instead of $(N/2) \log_2 N + N$, and we only pre-compute and store N powers of ψ_{2N}, instead of N powers of ψ_{2N} and N powers of ω_N.

If we straightforwardly calculate the INTT as [29], it will require $(N/2) \log_2 N$ modular multiplications and a post-processing step involving $2N$ modular multiplications. Similar to NTT, to eliminate the post-processing step, we merge it into the main INTT by replacing the original twiddle factor term $\omega_N^{-k_1}$ with $\psi_{2N}^{-(2k_1+k_2)}$ and evenly distributing the scaling factor N to each stage (in step 12 and 13). As a result, the CG INTT requires only $(N/2) \log_2 N$ modular multiplications instead of $(N/2) \log_2 N + 2N$, and we only pre-compute and store N powers of ψ_{2N}^{-1}, instead of N powers of ω_N^{-1} and N powers of ψ_{2N}^{-1}.

By merging the pre/post-processing steps into the NTT/INTT, the computational complexity and the memory occupation of twiddle factors are greatly reduced. However, the read/write structure of NTT and INTT are actually different, which means that we need to design different memory access patterns for NTT and INTT respectively. To solve this problem, we further unify the low-complexity INTT to NTT and make its structure consistent with that of NTT as Algorithm 1. Specifically, we first move the bit-reversal operation from the original last step of INTT to the first step as that of NTT. Next, a bit-reversal operation is performed on the index of twiddle factors to match the new butterfly structure of INTT (in step 10). Lastly, the input and output coefficient indexes of butterfly are changed to the same form as that of NTT (in step 12 and step 13). As a result, the read/write structure of INTT is the same as that of NTT, and we just design the data memory access pattern for NTT, while the access pattern for INTT is the same as that of NTT. We note

Algorithm 1 Unified Low-Complexity CG NTT/INTT

```
Let coefficient vector $\boldsymbol{a} = (a_0, \ldots, a_{N-1})$ and $\mathbf{A} = (A_0, \ldots, A_{N-1})$. Let
$\omega_N$ to be the primitive $N$-th root of unity and $\psi_{2N} = \omega_N^{1/2} \bmod q$.
Input: $\boldsymbol{a}$, $N$, $q$, $sel$, $\psi_{2N}^{i}$, $\psi_{2N}^{-i}$, where $i = 1, 2, \ldots, N-1$.
Output: $\mathbf{A} = NTT_{\psi_{2N}}^{N}(\boldsymbol{a})$ or $INTT_{\psi_{2N}^{-1}}^{N}(\boldsymbol{a})$
1: $\boldsymbol{a} \leftarrow$ BitReverse$(\boldsymbol{a})$
2: for $(s = 1; s \le \log_2 N; s = s + 1)$ do
3:   for $(j = 0; j < N/2; j = j + 1)$ do
4:     if $sel = 1$ then //NTT
5:       $k_1 = \left\lfloor j/2^{\log_2(N)-s} \right\rfloor \cdot 2^{\log_2(N)-s}$
6:       $k_2 = N/2^s$
7:       $A[j] \leftarrow a[2j] + a[2j+1] \cdot \psi_{2N}^{2k_1+k_2} \bmod q$
8:       $A[j+N/2] \leftarrow a[2j] - a[2j+1] \cdot \psi_{2N}^{2k_1+k_2} \bmod q$
9:     else //INTT
10:      $k_1 = \left\lfloor \text{BitReverse}(j)/2^{s-1} \right\rfloor \cdot 2^{s-1}$
11:      $k_2 = 2^{s-1}$
12:      $A[j] \leftarrow (a[2j] + a[2j+1]) \cdot (1/2) \bmod q$
13:      $A[j+N/2] \leftarrow (a[2j] - a[2j+1]) \cdot (\psi_{2N}^{-(2k_1+k_2)}/2) \bmod q$
14:    end if
15:  end for
16:  if $s \ne \log_2 N$ then
17:    $\boldsymbol{a} = \mathbf{A}$
18:  end if
19: end for
20: Return $(\mathbf{A})$
```

that [30] also presented a low-complexity NTT/INTT for CT algorithm (an in-place algorithm). But they need to design two different algorithms for NTT and INTT, respectively. More importantly, in order to make the memory access pattern of NTT adapt to INTT, they need to rearrange the inputs and outputs of INTT, which increases the complexity of controller and degrades the performance. Compared with them, the unified low-complexity CG NTT/INTT algorithm (an out-of-place algorithm) we presented completely unifies the algorithms and memory access patterns of NTT and INTT, avoiding the possible extra overheads caused by structure inconsistency of NTT and INTT.

B. Basis Extension and Scaling

The tensor products of BFV scheme requires tensoring the input ciphertexts and scaling by the factor t/q, followed by a rounding. In order to ensure the input ciphertexts are performed in integer domain, the tensor requires lifting the ciphertexts from ring R_q to R_Q by computing the ciphertexts on additional p residues. Let the polynomial (i.e., vector) $\mathbf{A}$ in R_q is represented in the RNS using the residues $\mathbf{A}_i \in R_{q_i}$ where each RNS base is a 32-bit primes q_i. The RNS basis extension is described as Algorithm 2. Suppose the moduli $q = \prod_{i=1}^{k} q_i$, $p = \prod_{j=k+1}^{k+k'} q_j$, $Q = q \cdot p$, where $k = 4$ and $k' = 5$ in our implementation. To avoid long integer arithmetic, we compute the simultaneous solution from RNS representation as follows.

$$\mathbf{A}_j \equiv \left(\sum (\mathbf{A}_i \cdot \tilde{q}_i \bmod q_i) \cdot q_i^* - \mathbf{V}' \cdot q\right) \bmod q_j \qquad (1)$$

Here $\mathbf{V}' = \lfloor \sum (\mathbf{A}_i \cdot \tilde{q}_i \bmod q_i)/q_i \rceil$, $q_i^* = q/q_i$ and $\tilde{q}_i = (q_i^*)^{-1} \bmod q_i$ after simplification, where $1 \le i \le 4$, $5 \le j \le 9$. Note that coefficients of $\mathbf{A}_i$, $\tilde{q}_i$ and q_i are all 32-bit

Algorithm 2 RNS Basis Extension

```
Let $\mathbf{A}_i \in R_{q_i}$, $q_i^* = q/q_i$, $\tilde{q}_i = (q_i^*)^{-1} \bmod q_i$, $q = \prod_{i=1}^{k} q_i$,
$p = \prod_{j=k+1}^{k+k'} q_j$, $Q = q \cdot p$, where $i = 1, \ldots, k$, $j =
k+1, \ldots, k+k'$.
Input: $\mathbf{A}_i, q_i, 1/q_i, \tilde{q}_i, q_i^*, q_j, q$.
Output: $\mathbf{A}_j = BasisExtension(\mathbf{A}_i)$.
1: for $(i = 1, i \le k, i = i + 1)$ do
2:     $\mathbf{A}_i' = \mathbf{A}_i \cdot \tilde{q}_i \bmod q_i$// Step 1
3: end for
4: for $(j = k + 1, j \le k + k', j = j + 1)$ do
5:     $\mathbf{A}_j' = \sum_{i=1}^{k} \mathbf{A}_i' \cdot q_i^* \bmod q_j$// Step 2
6:     $\mathbf{V}_j = \lfloor \sum_{i=1}^{k} \mathbf{A}_i' \cdot (1/q_i) \rceil \bmod q_j$// Step 3
7:     $\mathbf{V}_j' = \mathbf{V}_j \cdot q \bmod q_j$// Step 4
8:     $\mathbf{A}_j = \mathbf{A}_j' - \mathbf{V}_j' \bmod q_j$// Step 5
9: end for
10: Return $(\mathbf{A}_j)$
```

integers, the only source of error comes from the division by q_i. To ensure the correctness of division and rounding, we set the precision of the reciprocal q_i as 65-bit [13], [31]. One can bound the approximation error to 2^{-53} which has negligible impact on the correctness of RNS-BFV.

After complete the RNS basis extension, the ciphertext multiplication is performed in the RNS of R_Q in parallel. Then, the extended tensored ciphertexts need to be scaled down by t/q followed by a rounding operation to get an immediate scaled ciphertext $\lfloor (t/q) \cdot \mathbf{A}_i \rceil$, and a modular reduction by q_i is performed to get the RNS ciphertexts in R_{q_i}. The RNS basis scaling consists of two steps as described in Algorithm 3.

Algorithm 3 RNS Basis Scaling

```
Let $\mathbf{A}_i \in R_{q_i}$, $\mathbf{A}_j \in R_{q_j}$, $Q = q \cdot p$, $Q_i^* = Q/q_i$, $Q_j^* =
Q/q_j$, $\tilde{Q}_i = (Q_i^*)^{-1} \bmod q_i$, $\tilde{Q}_j = (Q_j^*)^{-1} \bmod q_j$, $I_i =
\text{Int}\{(t \cdot \tilde{Q}_i \cdot p)/q_i\}$, $R_i = \text{Real}\{(t \cdot \tilde{Q}_i \cdot p)/q_i\}$, where $i =
1, \ldots, k$, $j = k + 1, \ldots, k + k'$.
Input: $\mathbf{A}_i, \mathbf{A}_j, I_i, R_i, t, q_j, \tilde{Q}_j$.
Output: $\mathbf{CT}_i = BasisScaling(\mathbf{A}_i, \mathbf{A}_j)$.
1: for $(j = k + 1, j \le k + k', j = j + 1)$ do
2:     $\mathbf{S}_{I,j} = \sum_{i=1}^{k} \mathbf{A}_i \cdot I_i \bmod q_j$// Step 1
3:     $\mathbf{S}_{R,j} = \lfloor \sum_{i=1}^{k} \mathbf{A}_i \cdot R_i \rceil \bmod q_j$// Step 2
4:     $\mathbf{A}_j' = \mathbf{A}_j \cdot t\tilde{Q}_j q_j^* \bmod q_j$// Step 3
5:     $\mathbf{CT}_j = \mathbf{S}_{I,j} + \mathbf{S}_{R,j} + \mathbf{A}_j' \bmod q_j$// Step 4
6: end for
7: $\mathbf{CT}_i = BasisExtension(\mathbf{CT}_j)$// Step 5
8: Return $(\mathbf{CT}_i)$
```

In the first step, we compute the $\lfloor t/q \cdot \mathbf{A}_i \rceil \bmod q_j$ by the equation follows:

$$\lfloor t/q \cdot \mathbf{A}_i \rceil \bmod q_j = \lfloor \sum \mathbf{A}_i \cdot (t\tilde{Q}_i p/q_i) + \mathbf{A}_j \cdot (t\tilde{Q}_j q_j^*) \bmod q_j \rceil \bmod q_j \qquad (2)$$

Here $Q_i^* = Q/q_i$, $Q_j^* = Q/q_j$, $\tilde{Q}_i = (Q_i^*)^{-1} \bmod q_i$, $\tilde{Q}_j = (Q_j^*)^{-1} \bmod q_j$, $q_j^* = p/q_j$ for $1 \le i \le 4$, $5 \le j \le 9$. We pre-compute the integral and fractional parts of $(t \cdot \tilde{Q}_i \cdot p)/q_i$ with $I_i \in \mathbb{Z}_p$ and $R_i \in [-1/2, 1/2)$, in which R_i is stored as 65-bit to ensure the correctness of results [13], [31].

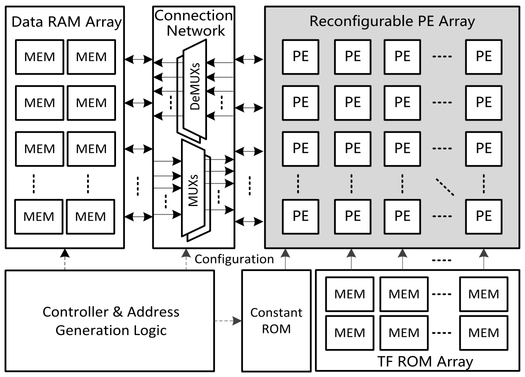

Fig. 1. Architecture of ReMCA.

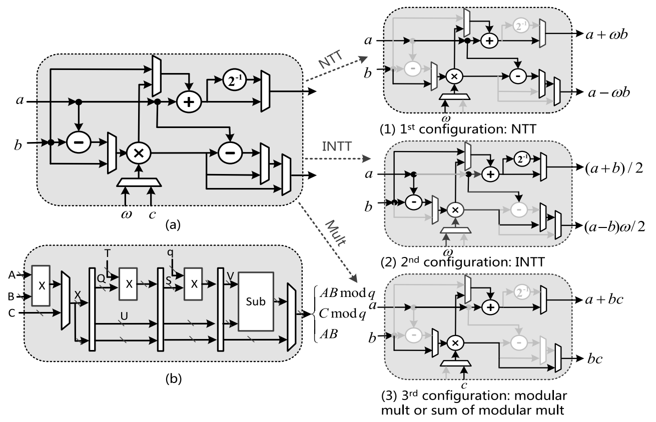
NTT
INTT
Mult
a
b
ω
c
(a)
(b)
A
B
C
T
q
Sub
AB mod q
C mod q
AB
a + ωb
a − ωb
(1) 1st configuration: NTT
(a+b)/2
(a−b)ω/2
(2) 2nd configuration: INTT
a + bc
bc
(3) 3rd configuration: modular mult or sum of modular mult

Fig. 2. Architecture of reconfigurable PE. (a) Reconfigurable PE, (b) Barrett modular multiplier.

In the second step, a basis switching from the RNS of p to the RNS of q is performed using the basis extension in Algorithm 2.

IV. Architecture

In this section, we investigate the hardware implementation and unified computing model of the ReMCA. By reusing the reconfigurable PE units in ReCMA, the building blocks such as basis extension/scaling and relinearization can reduce the resource occupation and improve the area efficiency.

A. Overall Architecture

The top level architecture of ReMCA is shown in Fig. 1. The ReMCA mainly consists of reconfigurable PE array, data memory array, twiddle factor (TF) memory array, constant memory and interconnection network. Among them, the reconfigurable PE array includes multiple homogeneous PE units which can be flexibly configured as butterfly units or the modular multipliers to perform the NTT/INTT or the modular multiplications in other computing units such as basis extension/scaling. The number of PEs and RNS channels can be flexibly adjusted according to the throughput (or performance) requirements of accelerator and the hardware resources of the FPGA chip. One row or multiple rows of PE can be configured as a channel to perform the polynomial arithmetic operations on one RNS base. For example, each row of PEs (e.g, eight PEs) can be configured as a channel for the medium-sized FPGA (e.g., Virtex-7 family), and every two rows (e.g., sixteen PEs) can be configured as a channel for the larger FPGA (e.g. UltraScale+ family). To accelerate the computation on different bases, multiple channels are further deployed in parallel. In order to maximum the overall utilization of PEs, we always restrict the number of PEs to be a power of two and a maximum of $N/2$ for each channel, where N refers to the number of points of NTT. Note that the PE array is mainly responsible for performing the homomorphic multiplication, while the homomorphic additions in each channel are performed by an additional modular adder.

The twiddle factors that associated with each channel (which corresponds to one RNS base) are divided into several subsets and stored in twiddle factor array, while the input polynomials, intermediate results and final results are stored in data memory array. Both the twiddle factors array and data memory array consist of multiple block-RAMs (BRAMs, i.e., MEM units). They can be access by multiple PEs in parallel to maximize the memory access bandwidth. The parameters that associated with the basis extension and scaling operations are pre-computed and stored in constant memory which is implemented by the distributed RAM (DRAM). In order to correctly transfer data between different BRAMs and PEs, a configurable interconnection network that consists of multi-level multiplexers and demultiplexers is implemented between data memory array and reconfigurable PE array.

In our implementation, we configure a total of 40 PEs, in which each row corresponds to one channel and each channel contains two slices and 8 PEs (4 for each slice). To maximize the parallelism of the processing path and match the number of extended RNS bases, we set five channels in PE array.

B. Reconfigurable PE Unit

1) The PE Unit: Though the conventional NTT unit supports both NTT and INTT operations, yet if the pre/post-processing steps are merged into the NTT and INTT, they will no longer support both of them and we need to design hardware for them separately. On the other hand, the modular multiplication and NTT/INTT require similar hardware architecture and they are used in different stages. Therefore, to have high area efficiency, a unified dynamic reconfigurable PE unit is presented in Fig. 2.

As one can see, the reconfigurable PE unit can be configured as NTT, INTT, modular multiplication or summation of modular multiplication by configuring the multiplexers. The architecture is fully pipelined to avoid long critical path in computations (there are total 13 pipelines in our PE). In consideration that implementing Montgomery modular multiplier requires more resources due to converting domain forth and back and demands more clock cycles, a Barrett modular multiplier with 11 pipelines for 32-bit modulus is deployed in PE unit. To accelerate the conditional subtraction step in the last stage of Barrett modular multiplier, we perform the conditional subtractions in the hardware implementations simultaneously by selecting the correct result using the carry bits of the subtraction results. Besides, in order to support the modular

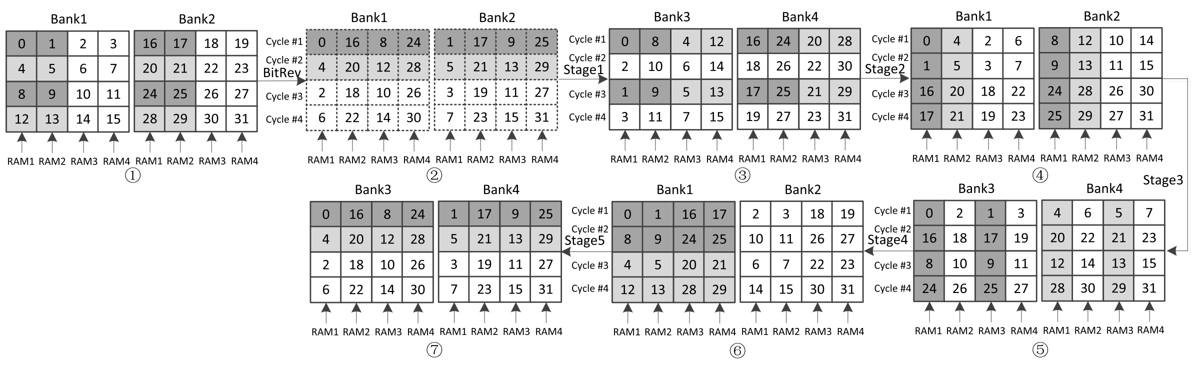

Fig. 3. Data memory access pattern of NTT for $N = 32,\ P = 4$.

multiplication of a 32-bit coefficient and a 65-bit constant in Step 3 of Algorithm 2 and Step 2 of Algorithm 3, we compute it using block multiplication, followed by a modular reduction operation. To achieve this, the Barrett modular multiplier can be flexibly configured as an integer multiplier or a separate modular reduction unit.

Additionally, to save the hardware resource occupation, we employ the resource sharing technique to implement modular adder and modular subtractor, we only use one modular adder and two subtractors in reconfigurable PE unit, which is one modular adder less than that of Sapphire [29]. On the other hand, according to Algorithm 1, the scaling factor N^{-1} is evenly distributed to each stage, thus we just multiply the polynomial coefficient by 1/2 in the addition path of INTT while the multiplication of 1/2 in the subtraction path can be merged into the twiddle factors. We employ more lightweight hardware to implement the 1/2 modular multiplication based on the following property.

Property 1: For a given $x/2 \bmod q$, when the modulus q is an odd prime, $1/2 \bmod q$ equals $(q+1)/2$, thus when x is an even number, $x/2 \bmod q$ equals to $(x \gg 1)$, while x is an odd number, $x/2 \bmod q$ can be expressed as

$$\begin{aligned}\frac{x}{2} &= \left(2\left\lfloor\frac{x}{2}\right\rfloor+1\right)\frac{q+1}{2} = \left\lfloor\frac{x}{2}\right\rfloor(q+1)+\frac{q+1}{2}\\ &= \left\lfloor\frac{x}{2}\right\rfloor+\frac{q+1}{2}(\bmod q) \qquad (3)\end{aligned}$$

where $\lfloor x/2\rfloor$ can be further expressed as $(x \gg 1)$. Therefore, the final result of $x/2 \bmod q$ can be chosen from $(x \gg 1)$ or $(x \gg 1) + (q+1)/2$, which is achieved only by a shifter, an adder and a multiplexer.

Compared with the butterfly unit in [30] that only supports NTT/INTT and modular multiplication with the fixed modulus, the reconfigurable PE we proposed not only supports the above-mentioned functions with the variable modulus, but also supports the summation of modular multiplication. Moreover, by merging the multiplicative factor 1/2 into the twiddle factors, the reconfigurable PE eliminates the multiplication of 1/2 in the subtraction path and improves the performance of PE unit. Lastly, the Barrett modular multiplier we presented employs a reconfigurable architecture and avoids the needs of other computing units for ReMCA.

2) Conflict-Free Memory Access for NTT/INTT: Designing efficient memory management is critical to avoid memory conflicts and achieve high throughput. We just need to design the memory access pattern for NTT due to the unified read/write structure of NTT/INTT we presented in Section III. To maximize the memory access bandwidth, we deploy four memory banks (i.e., *Bank1*∼*Bank4*) for *ping-pong* storing the polynomial coefficients. For 4096 degree polynomial and 4 PE units, each memory bank is composed of 4 RAMs with a depth of 512 and a width of 32 bits. Since each butterfly reads two inputs and writes two outputs in the same cycle, these RAMs are typically implemented using dual-port BRAMs.

Fig. 3 shows the data memory access pattern of NTT for 32-degree polynomial and four PE units. Initially, the input polynomial coefficients are split among *Bank1* and *Bank2* on the basis of the least and most significant bits of the coefficient index, i.e., the first half are stored in *Bank1*, while the other half are stored in *Bank2*. The coefficients with even index are stored in RAM1 and RAM3, and the odd index coefficients are stored in RAM2 and RAM4. Note that before performing NTT, a bit-reversal operation is required. However, the bit-reversal operation is typically time-consuming or memory-consuming. To avoid this expensive operation, we only change the address mapping pattern. Specifically, in Step 1, all the coefficients are stored in *Bank1* and *Bank2* in normal order. Next, if we perform bit-reversal on the input coefficients, all the coefficients are stored in bit-reversal order as Step 2, so the coefficient pairs in the first row of Step 2 can be processed by four PEs in parallel during the first stage of NTT. But we find that the coefficient pairs in the first row of *Bank1* and *Bank2* in Step 2 come from the first and *N/16*-th (i.e., 1/2 depth of RAM) row of the first two RAMs of *Bank1* and *Bank2* in Step 1 respectively, which can be read concurrently by four PEs using dual-port pattern. Similarly, the coefficient pairs in the following rows in Step 2 come from different positions of (the first or the last) two RAMs in Step 1 and can be accessed by all four PEs in parallel. Thus, by modifying the address mapping method of input coefficients, we eliminate the separate bit-reversal step and just directly perform the NTT using the coefficient pairs from Step 1 as the processing sequence of Step 2 (the storage depicted in Step 2 actually does not exist).

After completing the bit-reversal permutation, all of the coefficients start to perform the NTT. For CG algorithm, the coefficients of the polynomial are accessed in the same order for each stage. Take stage 1 as an example, the PEs first read the coefficients in the first row of *Bank1* and *Bank2* in the first cycle (as Step 2 shows), the transformed coefficients are stored in the first half and last half of RAM1 and RAM2 in *Bank 3* and *Bank 4*, respectively. Then, in the next cycle, the PEs read the coefficients in the second row of *Bank1* and *Bank2*, and the transformed coefficient are stored in the first half and last half of RAM3 and RAM4 in *Bank 3* and *Bank 4*, respectively. The memory access patterns for the remaining rows and stages are similar. Since the PEs employ full pipeline architecture, the memory read and write are also run in full pipeline and *ping-pong* manner between *Bank1, 2* and *Bank 3, 4*, so they achieve the throughput of $E = 8$ elements/cycle.

A conflict-free memory access pattern was also introduced in [20] and [21]. However, the CT NTT algorithm they employed is an in-place algorithm, its memory access pattern will change with the number of PEs. For the multi-core architecture, it is not conducive to realization of scalability. In contrary, the CG NTT algorithm we employed is an out-of-place algorithm, and the memory access pattern of each stage always keeps the same regardless of the number of PEs. Therefore, it has better scalability for the multi-core architecture. Moreover, by changing the address mapping pattern of dual-port memory, we completely avoid the bit-reversal operation, the clock cycles or memory overheads are greatly reduced.

C. Unified Computing Model

1) Unified Hardware Architecture Mapping Model: By analyzing the data flow graph (DFG) of RNS-BFV scheme, we find that there is usually no data dependence between computation steps of homomorphic evaluations. Hence, in order to accelerate the computations, we divide the PE units of each channel into two slices, i.e., SliceA and SliceB, while each slice is composed of four PE units. According to different computational requirements, we conclude four different mapping modes. Fig. 4 shows the unified hardware architecture mapping model of ReMCA.

Among them, Mode 1 is used to compute the 32-bit modular multiplications of four contiguous integers in vector $\mathbf{A}_i$ and four contiguous integers in vector $\mathbf{B}_i$ or four constants in parallel, i.e., each PE in a slice is responsible for computing one element of the product of two vectors. Thus, the point-wise multiplication of two 4096-degree polynomial coefficients is computed by a slice in 1024 clock cycles (cc). Mode 2 is used to compute the summation of four products, while the inputs of four products are from four different vectors and four constants respectively. i.e., each PE is responsible for computing a product while each slice is responsible for computing a summation of four products by reusing the modular adder of PE. Note that the Step 2 in Algorithm 2 and Step 1 in Algorithm 3 can be mapped as Mode 2, and for a 4096-degree polynomial, the summation of four products requires 4096 cc. The NTT/INTT transforms are computed using Mode 3, i.e., each NTT/INTT is computed by a slice, hence, performing a NTT/INTT requires a total of $(N/2P)\log_2 N = 6144$ cc for a 4096-degree polynomial. Different from former three modes, Mode 4 requires the cooperation of first four channels. It is

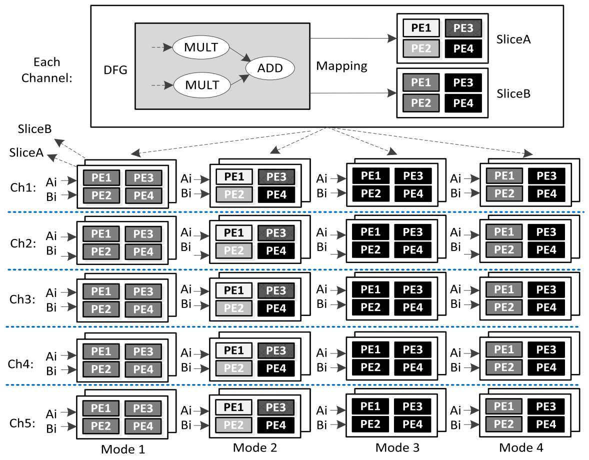

Fig. 4. Unified hardware architecture mapping model of ReMCA.

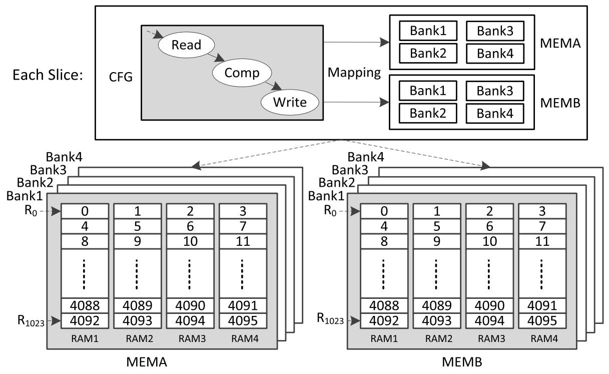

Fig. 5. Unified data memory organization model of ReMCA.

used to compute the summation of four products followed by a rounding operation as shown in Step 3 in Algorithm 2 and Step 2 in Algorithm 3, i.e., the inputs of a product come from a 32-bit integer vector and a 65-bit constant, so every two PEs in a slice is responsible for computing a product using the block multiplication, and the summation of products is computed by reusing the homomophic addition unit. As a result, for four 4096-degree polynomials, the slices in the first four channels can compute all computations in 2048 cc, while the remaining channel 5 is in the idle state. In four mapping modes, there is no data communication between various PEs in Mode 1, 2 and 3, and only the PEs in Mode 4 need data communication. In Mode 4, one of the PEs is used to compute the partial product of the 32-bit integer and the higher 32-bit part of the constant, while the other one is used to compute the product of the integer and the remaining parts. Then, the former result is left shifted by 32-bit and added to the latter one to get the final result.

2) Unified Data Memory Organization Model: Similarly, by analyzing the *Control Flow Graph* (CFG) and algorithm of RNS-BFV, we present the unified data memory organization model of ReMCA as shown in Fig. 5.

According to memory access requirements, we divide the memory space of each slice into two parts: MEMA and MEMB. Among them, MEMA consists of four memory banks, where each memory bank further contains four 1024-depth and 32-bit-width dual-port RAMs. It is used to store the inputs/outputs and intermediates results of almost all

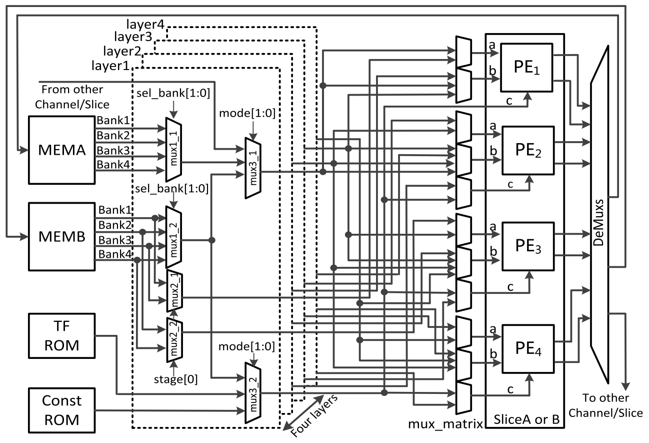

Fig. 6. Detailed structure of interconnection network of each slice.

functional units in homomorphic evaluation of RNS-BFV except for the NTT and INTT. The memory organization of MEMB is the same as that of MEMA. But MEMB is mainly used to store the inputs/outputs and intermediate results of NTT and INTT. Thus when it is used to perform the NTT/INTT, the transformed data only occupies the first 512 positions of each bank, and the last 512 positions can be used to store other temporary data. When ReMCA does not perform NTT/INTT, MEMB can also be used to store the intermediate results or temporary data.

In order to better clarify the whole architecture in terms of data scheduling between memories and PEs, we present the detailed structure of the interconnection network of each slice of ReMCA as shown in Fig. 6. Because there are four banks in each MEM and each bank contains four RAMs, in order to output the data from four different RAMs (of the same bank) in parallel, we deploy four layers multiplexers, i.e., one layer is responsible for outputting the data of one RAM. For example, when the slice is working at Mode 1 and the control signals of mux1_1 and mux1_2 are set to 2'b00, four data are read from RAM1-RAM4 of *Bank1* of MEMA or MEMB in parallel. The mux2_1 and mux2_2 are used to output the data from *Bank1* or *3* and *Bank2* or *4* respectively for performing the NTT or INTT. In addition, the mux3_1 is used to output the data from the MEMA or MEMB in this channel or the other channel or slice, while the mux3_2 is used to output the constants or twiddle factors from corresponding ROMs, or output the data from MEMB according to mapping modes. When all the data are output from four layers, they are distributed into different PEs through the mux_matrix. Lastly, the outputs of PEs are written back to MEMA or MEMB in this channel or other channel or slice through the demultiplexers. Since the PEs only perform NTT/INTT or basic modular arithmetic operations, the number of outputs is always not greater than that of inputs, so it will not cause the expansion of intermediate data, nor it will cause the data congestion, Therefore, the overhead generated by the intermediate write-back operations is relatively small.

When the slice is configured as Mode 1, four 32-bit data of different RAMs in the same bank are read from MEMA or MEMB through mux1_1 and mux3_1, and the constants

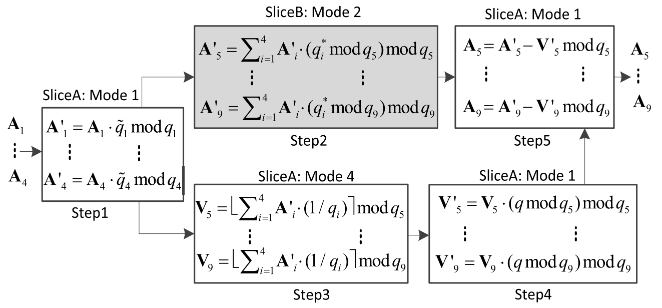
SliceB: Mode 2
SliceA: Mode 1
SliceA: Mode 1
SliceA: Mode 4
SliceA: Mode 1
$\mathbf{A}_1 \vdots \mathbf{A}_4$
$\mathbf{A}'_1 = \mathbf{A}_1 \cdot \tilde{q}_1 \bmod q_1 \;\vdots\; \mathbf{A}'_4 = \mathbf{A}_4 \cdot \tilde{q}_4 \bmod q_4$
Step1
$\mathbf{A}'_5 = \sum_{i=1}^{4} \mathbf{A}'_i \cdot (q_i^* \bmod q_5) \bmod q_5 \;\vdots\; \mathbf{A}'_9 = \sum_{i=1}^{4} \mathbf{A}'_i \cdot (q_i^* \bmod q_9) \bmod q_9$
Step2
$\mathbf{V}_5 = \lfloor \sum_{i=1}^{4} \mathbf{A}'_i \cdot (1/q_i) \rceil \bmod q_5 \;\vdots\; \mathbf{V}_9 = \lfloor \sum_{i=1}^{4} \mathbf{A}'_i \cdot (1/q_i) \rceil \bmod q_9$
Step3
$\mathbf{V}'_5 = \mathbf{V}_5 \cdot (q \bmod q_5) \bmod q_5 \;\vdots\; \mathbf{V}'_9 = \mathbf{V}_9 \cdot (q \bmod q_9) \bmod q_9$
Step4
$\mathbf{A}_5 = \mathbf{A}'_5 - \mathbf{V}'_5 \bmod q_5 \;\vdots\; \mathbf{A}_9 = \mathbf{A}'_9 - \mathbf{V}'_9 \bmod q_9$
Step5
$\mathbf{A}_5 \vdots \mathbf{A}_9$

Fig. 7. Mapping method of basis extension unit.

are read from the Constant ROM (or the data from MEMB) through mux3_2, then both of them are send to port *b* and *c* of different PEs, respectively. When the slice is configured as Mode 2, four 32-bit data are read from MEMA or MEMB of this channel or other channel/slice through mux1_1 and mux3_1. Then they are sent to port *b* and *c* of different PEs together with the constants read from Constant ROM. The case of Mode 4 is similar to that of Mode 1. The only difference is the data of four RAMs are divided two parts, and each part (i.e., two data) is shared by four PEs each time. When the slice is configured as Mode 3, eight 32-bit data of *Bank1, 2* or *Bank3, 4* are read from MEMB through mux2_1 and mux2_2 of four layers, they are sent to port *a*, *b* and *c* of different PEs together with the twiddle factors read from TF ROM.

V. Mapping Method and Execution Flow

The homomorphic multiplication of RNS-BFV includes four computing units: basis extension, ciphertext multiplication, basis scaling and relinearization, next, we introduce how these computing units are mapped on ReMCA efficiently.

A. Computing Units Mapping

1) Basis Extension Unit: Fig. 7 shows the mapping method of basis extension unit. The first step involves four 32-bit modular multiplications. So we map these modular multiplications on the SliceA of first four channels with the Mode 1 respectively. i.e., SliceA of each channel is responsible for computing a modular multiplication. Step 2 requires computing five summation-of-products while each summation further involves four products. To speed up it, we separately map it on the SliceB of all five channels with the Mode 2. Considering that Step 3 and Step 4 are independent with Step 2, after ReMCA completes Step 1, the SliceA in all channels are in the idle state, thus Sep 3 and Step 4 can be mapped on SliceA of each channel in serial. In Step 3 the summation of four products followed by a rounding operation is performed. Note that the summation of four products only needs to be computed once and the modular reductions of q_j can be performed by reusing the homomorphic addition units. Thus, only first four channels are mapped with Mode 4 and the last one is in the idle state. On the other hand, though the constant $1/q_i$ are stored in ROM with a precision of 65 bits after the decimal point, the first 30 bits after the decimal point are actually all-zeros. Hence, the multiplication is actually computed between a 32-bit coefficient vector and 35-bit non-zero bits of $1/q_i$.

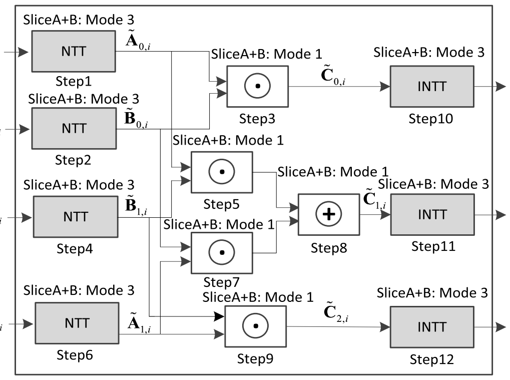

Fig. 8. Mapping method of ciphertext multiplication unit.

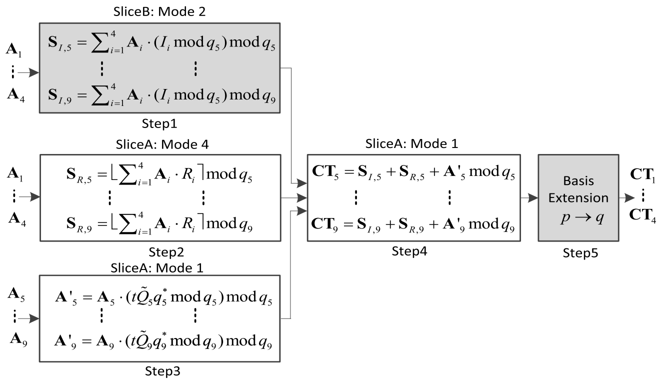

Fig. 9. Mapping method of basis scaling unit.

In Step 4, the modular multiplications of coefficient vector and constants $q \bmod q_j$ can be mapped in all five channels with Mode 1. Lastly, the modular subtraction between results of Step 2 and Step 4 takes almost no additional cycles, thus we directly merge Step 5 into Step 4 by reusing the modular subtractor in each PE.

2) Ciphertext Multiplication Unit: After performing the basis extension, the number of bases is extended from original four to nine, and the corresponding ciphertext polynomials become $\mathbf{A}_{0,i}$, $\mathbf{A}_{1,i}$, $\mathbf{B}_{0,i}$ and $\mathbf{B}_{1,i}$, where $1 \le i \le 9$. So next we perform the ciphertext multiplications of these residual polynomials in parallel on extended bases. Fig. 8 shows the mapping method of ciphertext multiplication unit. Since the number of bases is almost twice the number of channels, we map the operations related to the first five bases on the SliceA of all five channels, and map the operations related to the last four bases on the Slice B of first four channels, respectively, i.e., keep the SliceB of the last channel in the idle state. Besides, we map all the NTTs and INTTs on ReMCA with Mode 3, while the other operations such as point-wise multiplications are mapped as Mode 1.

Instead of the conventional method which requires two separate INTTs on $\tilde{\mathbf{A}}_{0,i} \cdot \tilde{\mathbf{B}}_{1,i}$ and $\tilde{\mathbf{A}}_{1,i} \cdot \tilde{\mathbf{B}}_{0,i}$ to convert them to coefficient domain and then perform the modular addition of them, we directly perform the modular addition of $\tilde{\mathbf{A}}_{0,i} \cdot \tilde{\mathbf{B}}_{1,i}$ and $\tilde{\mathbf{A}}_{1,i} \cdot \tilde{\mathbf{B}}_{0,i}$ in NTT domain and convert the summed result to the coefficient domain using only one INTT, i.e., one INTT less than conventional method in each channel. By this way, the performance of ReMCA can be significantly improved due to the multi-channel architecture. On the other hand, the modular addition in Step 8 costs almost no cycles, thus we merge it into the point-wise multiplication of Step 7, i.e., the summation of point-wise multiplication is performed. The processing sequence of each functional unit is vital to the memory utilization. In order to minimum the on-chip memory occupation, we perform the ciphertext multiplication according to the steps as Fig. 8 shows. We first perform the NTTs of $\mathbf{A}_{0,i}$ and $\mathbf{B}_{0,i}$, and compute the product of them. Then, we perform the NTT of $\mathbf{B}_{1,i}$ and compute the product of $\mathbf{A}_{0,i}$ and $\mathbf{B}_{1,i}$. At this time, the storage of $\mathbf{A}_{0,i}$ is no longer used, it can be overlapped by the product of $\mathbf{A}_{0,i}$ and $\mathbf{B}_{1,i}$, thereby saving the memory occupation. A similar strategy also applies to the other steps.

3) Basis Scaling Unit: Next, we map the basis scaling unit on the ReMCA as depicted in Fig. 9. From the flow diagram we see that the operations of Step 1, Step 2 and Step 3 are independent of each other, and the Step 1 is the most expensive operation since it involves five summation-of-products. Similar to basis extension unit, we map the operations in Step 1 on the SliceB of all five channels with the Mode 2. Step 2 only involves the summation of four products, followed by a rounding operation. Thus, we map Step 2 on the SliceA of first four channels with Mode 4, while the last one channel is in the idle state. Similarly, to speed up the computation, each product is computed between a 32-bit coefficient vector and 35-bit non-zero bits of R_i. After ReMCA completes the computations of Step 2, the SliceA in each channel are in the idle state, thus we continue to map the modular multiplications in Step 3 and modular additions in Step 4 on the SliceA of all five channels with Mode 1 Once the computation of Step 4 is finished, Step 5 will receive five residues in the RNS of p, thus it continues to reuse the basis extension unit of Fig. 7 to compute the resides in the RNS of q.

4) Relinearization Unit: Relinearization is another important unit for homomorphic multiplication. Let the RNS ciphertexts of relinearization as $\mathbf{C}^*_{0,i}$, $\mathbf{C}^*_{1,i}$ and $\mathbf{C}^*_{2,i}$, where $1 \le i \le 4$. Instead of decomposing the ciphertexts $\mathbf{C}^*_2$ into bit or digits as past, we decompose $\mathbf{C}^*_2$ into its RNS components $\mathbf{C}^*_{2,i}$ to make the relinearization compatible with RNS. Specifically, we first choose a uniform $\boldsymbol{\alpha}_j \in R_q$ and $\mathbf{e}_j \leftarrow \chi_\sigma$, and set $\boldsymbol{\beta}_j = \tilde{q}_j q^*_j \mathbf{s}^2 - \boldsymbol{\alpha}_j \mathbf{s} + \mathbf{e}_j \bmod q$ for each $1 \le j \le 4$. Thus, the relinearization keys consist of the vectors $\mathbf{W}_j = (\boldsymbol{\beta}_j, \boldsymbol{\alpha}_j)$. To compute the relinearization ciphertexts, we first compute $\tilde{\mathbf{C}}_{0,i} = \sum_{j=1}^{4} \left[[\boldsymbol{\beta}_j]_{q_i} \cdot \mathbf{C}^*_{2,j} \right]_{q_i}$ and $\tilde{\mathbf{C}}_{1,i} = \sum_{j=1}^{4} \left[[\boldsymbol{\alpha}_j]_{q_i} \cdot \mathbf{C}^*_{2,j} \right]_{q_i}$, and then compute $\mathbf{C}^\times_{0,i} = [\mathbf{C}^*_{0,i} + \tilde{\mathbf{C}}_{0,i}]_{q_i}$ and $\mathbf{C}^\times_{1,i} = [\mathbf{C}^*_{1,i} + \tilde{\mathbf{C}}_{1,i}]_{q_i}$.

The NTT transforms are extremely time-consuming. If we employ the conventional method that performing the NTTs of relinearization keys online (i.e. on FPGA), it will increase four more NTTs for each $\boldsymbol{\beta}_j$ and $\boldsymbol{\alpha}_j$, respectively. Even all the NTTs are mapped on SiceA and SliceB with Mode 3 in parallel, there will be more than 52% performance degradation for the relinearization operation. Therefore, in order to speed up the relinearization operation, we pre-compute the relinearization keys and convert them to NTT domain offline

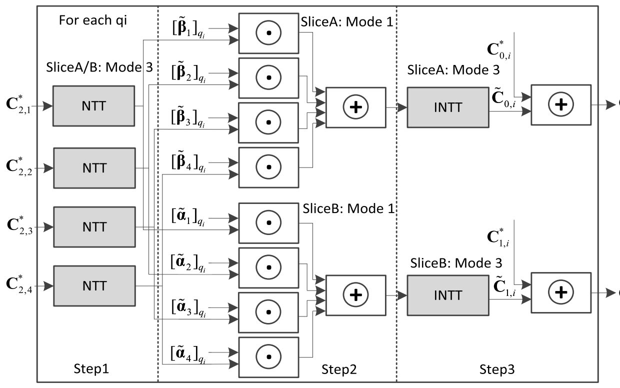
For each qi
SliceA/B: Mode 3
$\mathbf{C}^*_{2,1}$
$\mathbf{C}^*_{2,2}$
$\mathbf{C}^*_{2,3}$
$\mathbf{C}^*_{2,4}$
NTT
$[\tilde{\boldsymbol{\beta}}_1]_{q_i}$
$[\tilde{\boldsymbol{\beta}}_2]_{q_i}$
$[\tilde{\boldsymbol{\beta}}_3]_{q_i}$
$[\tilde{\boldsymbol{\beta}}_4]_{q_i}$
$[\tilde{\boldsymbol{\alpha}}_1]_{q_i}$
$[\tilde{\boldsymbol{\alpha}}_2]_{q_i}$
$[\tilde{\boldsymbol{\alpha}}_3]_{q_i}$
$[\tilde{\boldsymbol{\alpha}}_4]_{q_i}$
SliceA: Mode 1
SliceB: Mode 1
$\mathbf{C}^*_{0,i}$
$\mathbf{C}^*_{1,i}$
SliceA: Mode 3
SliceB: Mode 3
INTT
$\tilde{\mathbf{C}}_{0,i}$
$\tilde{\mathbf{C}}_{1,i}$
$\mathbf{C}^\times_{0,i}$
$\mathbf{C}^\times_{1,i}$
Step1
Step2
Step3

Fig. 10. Mapping method of RNS based relinearization unit.

(i.e., on CPU) during the key generation phase. The mapping method of relineaization is shown in Fig. 10. Because there are only four bases, all the steps in relinearization unit can be mapped on either SliceA or SliceB of the first four channels. To be specific, in Step 1, the NTTs of $\mathbf{C}^*_{2,j}$ for q_1 and q_2 can be mapped on SliceA and SliceB of first four channels with Mode 3, respectively, while the NTTs for q_3 and q_4 are mapped in the next stage with the similar manner. In Step 2, two groups of modular multiplications on q_1 are mapped on SliceA and SliceB of each channel with Mode 1 respectively, while the modular multiplications for other $q_i s$ are mapped in the following stages with the similar manner. To reduce the computational complexity, the modular additions are also merged into the Step 2. In the last step, the INTT transforms of $\tilde{\mathbf{C}}_{0,i}$ and $\tilde{\mathbf{C}}_{1,i}$ for each q_i are mapped on SliceA and SliceB of each channel with Mode 3 respectively. Similarly, the modular additions at follows are also merged into the INTT steps.

B. Execution Flow of RNS-BFV

To show the algorithm mapping and processing flow more clearly, we list the overall execution flow of homomorphic multiplication of RNS-BFV on ReMCA in Fig. 11. Above four computing units consume most of the computational complexity and can be accelerated on ReMCA. The ReMCA can achieve parallel processing both at space and time dimension. At space dimension, multiple RNS channels and multiple PEs are organized in parallel to accelerate the computation. The number of PEs and RNS channels can be flexibly adjusted according to the different performance requirements of accelerator and the hardware resources of FPGA. In our implementation, there five parallel RNS channels are configured, while each channel further contains two four-PE slices. At time dimension, to compute multiple sequential steps in each computing unit, the PEs employ the circuit-level pipelines, and different steps of the kernel codes can be computed by calling the four-PE slices of each channel repeatedly. By reusing the highly unified reconfigurable multi-core architecture, ReMCA achieves the maximum resource utilization, and the area efficiency is greatly sped up.

VI. Implementation Results and Comparisons

A. FPGA Implementation Result

We developed ReMCA into Verilog module and synthesized it using Xilinx Vivado tool for the Virtex-7 XC7VX1140T FPGA family. The synthesis results of ReMCA, PE ‘and Barrett modular multiplier are summarized in Table I. Note that in each channel, we deployed an additional modular adder to perform the homomorphic addition operations, its hardware resource and latency are also included in that of ReMCA. We synthesized the design and achieved 250MHz frequency under the parameter set ($N = 4096$, $\log(q) = 128$-bit, $\log(q_i) = 32$-bit).

By analyzing the implementation result, we find that each PE occupies 10 DSPs and there are a total of 40 PEs in ReMCA, therefore, the DSP occupation of ReMCA is 400. For the data memory, we deploy two MEMs in each slice, thus the total number of BRAMs in all five channels (each channel contains two slices) is 320. For the twiddle factors memory, the number of BRAMs in each slice is eight in total, but the SliceB in the last channel is always in the idle state when performing the NTT/INTT, thus there are total 72 BRAMs. As a result, ReMCA occupies a total of 392 BRAMs. From the table, it can be seen that the design is constrained on the memory size.

TABLE I
FPGA Implementation Result of ReMCA on XC7VX1140T

***N*=4096**	Slice LUTs	Slice Registers	RAMB36E1	DSP48E1
Total available	712000	1424000	1880	3360
ReMCA	46591 (6.54%)	35551 (2.50%)	392 (20.85%)	400 (11.90%)
Each PE	1043 (0.15%)	811 (0.06%)	– –	10 (0.30%)
Each Barrett	560 (0.08%)	553 (0.04%)	– –	10 (0.30%)

TABLE II
Timing Results of Primitive Operations

Operation	**Speed**	
	(cycles)	**(μs)**
Basis Extension	16384	65.54
Ciphertext Mult	47104	188.42
Basis Scaling	24576	98.30
Relinearization	22528	90.11
Total	110592	442.37
Each NTT	6144	24.58
Each INTT	6144	24.58
Each Point-wise Mult	1024	4.10

Table II shows the timing results of primitive operations in ReMCA. Follow the algorithm flow, the basis extension first lifts a polynomial from R_q to R_Q in 65.54 μs. The SliceA and SliceB in each channel can work in parallel. The ciphertext multiplication computes the modular multiplications on the extended bases in 188.42 μs. It is the most expensive operation in homomorphic evaluation. The basis scaling operation first computes the intermediate results in the RNS of p. Then it repeats the basis extension to map this result to the RNS of q. Hence, its computational time is larger, up to 98.30 μs. The relinearization reduces the dimension of each RNS ciphertext from three to two in 90.11 μs. By employing the RNS-based relinearization, ReMCA completely avoid converting the ciphertext forth and back from polynomial form to RNS representation. Finally, we can compute 2260 homomorphic multiplications per second.

We put significant efforts in reducing the computational complexity of NTT/INTT. By merging the pre/post-processing

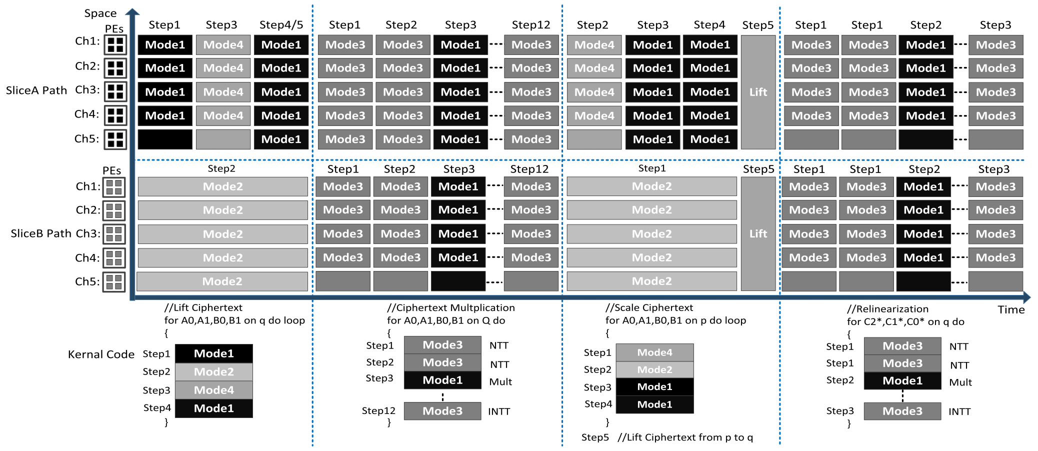
Space
PEs
Ch1:
Ch2:
SliceA Path Ch3:
Ch4:
Ch5:
SliceB Path
Step1
Step2
Step3
Step4/5
Step4
Step5
Step12
Mode1
Mode2
Mode3
Mode4
Lift
Time
//Lift Ciphertext for A0,A1,B0,B1 on q do loop
//Ciphertext Multplication for A0,A1,B0,B1 on Q do
//Scale Ciphertext for A0,A1,B0,B1 on p do loop
//Relinearization for C2*,C1*,C0* on q do
Kernal Code
NTT
Mult
INTT
Step5 //Lift Ciphertext from p to q

Fig. 11. Overall execution flow of RNS-BFV on ReMCA.

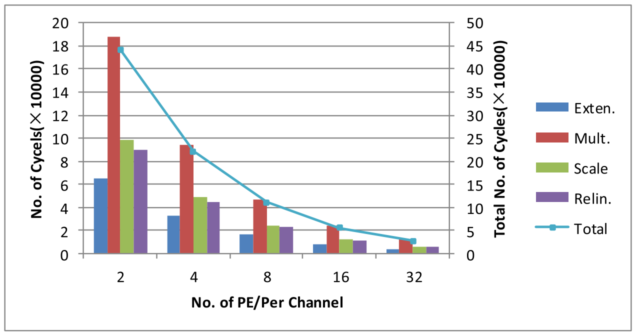

Fig. 12. The relationship between the number of PEs and performance.

into the NTT/INTT and introducing the reconfigurable PE array architecture, the computational time of NTT/INTT and point-wise multiplication on each slice is reduced to 24.58 μs and 4.10 μs, respectively.

We also provide the relationship between the number of PEs and the performance of ReMCA for the polynomials with 4096 degrees and 32 bits small moduli in Fig. 12. We find that with the doubling of the number of PEs, the latency (No. of cycles) of basis extension, cipertext multiplication, basis scaling and relinearization are reduced by nearly half, respectively, which means the performance of ReMCA is almost doubled. However, although the increase of the number of PEs in per channels will greatly improve the performance of ReCMA, the hardware resource overhead becomes an obstacle for the resource constraint FPGA. Therefore, for most of FPGAs, a medium number of PEs is a more common choice. In addition, by analyzing the cycle ratio of different operations, we also conclude that the ciphertext multiplication is the most time-consuming because it contains more NTTs and INTTs.

B. Comparison of NTT/INTT Acceleration

According to Table II, we find that more than 63% of the clock cycles are spent in ciphertext multiplication and relinearization. At lower arithmetic level, more than 56% of the clock cycles are spent in NTT or INTT. Hence, we first compare the performance of NTT/INTT of ReMCA with the related works as shown in Table III. For fair comparison, we fix the bit-length of each small moduli as 32 bits and compare with the related works with the same or less modulus bit-length. As the FPGA has various types of hardware resources, the Area-Time-Products (ATPs) are measured and compared by multiplying time by the number of LUTs, FFs, BRAMs and DSPs respectively.

Mert *et al.* [15] proposed a four-step NTT hardware with a 30-bit modulus to accelerate the polynomial multiplication of BFV scheme. For comparison, we provide the implementation result of one slice (which corresponds to four PEs and a 32-bit modulus) in Table III. In order to improve the performance of NTT, they deployed a large number of modular multipliers (more than 50). Besides, in order to guarantee that there is only one subtraction at the end of modular reduction, they restricted the bit-width of moduli between 22-bit and 32-bit. As a result, the throughput of them is 10.69× higher than that of ReMCA. But even so, the ATPs measured by LUT and DSP of ReMCA are improved by 1.1× and 1.41×, respectively. This is because the NTT and INTT of [15] require separate pre/post-processing steps which increase the computational overhead, while ReMCA completely merges them into NTT/INTT. Additionally, it is found that the number of DSP of [15] is 14.98× more than that of ReMCA, if more DSPs are deployed, the performance of ReMCA can be further improved.

Cathébras *et al.* [14] presented a hardware implementation of a RPM to accelerate the RNS-BFV scheme. To reduce the memory overhead of twiddle factors, they proposed a twiddle factor generator which can generate the twiddle factors on-the-fly. But this also brings additional hardware resource overhead. Though they did not provide the concrete latency of NTT, we conservatively estimate the latency of NTT occupies at least 50% of that of polynomial multiplier based on our implementation result (56%) and their profiling of HPS (60%) [13]. Compared to their work with eight RPMs and four 30-bit small moduli, ReMCA with 40 PEs and four 32-bit small moduli achieves 6.51× throughput improvement, while the ATPs measured by LUT, FF, BRAM and DSP are improved by 5.48×, 8.78×, 2.26× and 5.54×, respectively.

Öztürk *et al.* [32] presented an FPGA architecture to accelerate LTV scheme [33]. To accelerate large polynomial multiplication and support homomorphic evaluation of Prince and AES algorithms, they use CRT to convert the input

TABLE III
PERFORMANCE COMPARISON OF NTT/INTT

Design	**Platform**	***N***	***$\log q_i$* (bit)**	**No. bases**	**Freq (MHz)**	**Cycles**	**Time (μs)**	**Thr† (MB/s)**	**LUT /ATP††**	**FF /ATP††**	**BRAM /ATP††**	**DSP /ATP††**
Mert [15]	Virtext-7	4096	30	1	200	–	2.3	6793.5	70000 /0.16	– /–	129 /0.30	559 /1.38
ReMCA	Virtext-7	4096	32	1	250	6144	24.58	635.68	5968 /0.15	4394 /0.11	40 /0.98	40 /0.98
Cathébras [14]	Virtext-7	4096	30	4	200	–	150	390.63	41964 /6.30	50961 /7.64	147 /22.05	363 /54.45
ReMCA	Virtext-7	4096	32	4	250	6144	24.58	2542.72	46591 /1.15	35551 /0.87	392 /9.74	400 /9.83
Öztürk [32]	Virtext-7	32768	32	41	250	–	2086.9	2551.02	219192 /457.43	90789 /189.47	193 /402.77	768 /1602.74
ReMCA	Virtext-7	32768	32	41	250	61440	245.76	20853.68	194084 /47.70	153050 /37.61	1759 /432.29	1680 /412.88

†Throughput (Thr) = N×(No. of bits (q))/Time. ††ATP=No. LUT or FF or BRAM or DSP multiply the total time (second (column #11, #12) or millisecond (column #13, #14));

polynomials into the residue polynomials, and set the maximum number of RNS bases and modular multipliers as 41 and 256, respectively. As our parameter set is much less than theirs, to compare with them more fairly, we extend the polynomial degree and the number of bases of ReMCA to the same size as them. We set a total of 42 channels in extended ReMCA, each channel contains only one slice, and each slice is composed of four PEs (i.e., 168 PEs in total). In comparison, the throughput of extended ReMCA is 8.17× higher benefiting from highly unified multi-channel multi-PE architecture. Meanwhile, ATPs measured by LUT, FF and DSP are sped up by 9.59×, 5.04× and 3.88×, respectively. Though the number of BRAMs and corresponding ATP of extended ReMCA are 9.11× and 1.07× that of [32], this is because the intermediate results and related parameters of ReMCA are stored on the board, while [32] transports them from CPU.

C. Comparison of BFV Acceleration

Table IV shows the comparisons of homomorphic evaluation of BFV scheme. In order to compare with the works of Roy *et al.* [17] and Turan *et al.* [18], which have the common ciphertext parameter (the bit-length of each small moduli is 30-bit and the number of bases is seven), we extend ReMCA to a new design by adding two additional channels, i.e., there is total seven channels, each channel contains two slices, and each slice contains four PEs (i.e., a total of 56 PEs and seven 32-bit small moduli). Note that the Virtex UltraScale+ FPGA in [18] and the Zynq UltraScale+ FPGA in [17] are both manufactured using 16 nm FinFET process. Based on the synthesis results of ReMCA on these two FPGAs, it shows that their performance is almost the same. Therefore, we only provide the result of the extended ReMCA on Zynq UltraScale+ FPGA for comparison. As can be seen that the frequency of the extend ReMCA is improved by 1.52× on the new platform.

As discussed before, a RNS-BFV scheme was accelerated by Cathébras *et al.* [14]. Benefit from less storage space of twiddle factors, the number of BRAMs and ATP measured by BRAM of [14] (with eight RPMs and four 30-bit small moduli) are 1.73× and 1.42× less than that of ReMCA (with 40 PEs and four 32-bit small moduli), respectively. But even so, the throughput of ReMCA is improved by 1.45×, while the ATPs measured by LUT, FF and DSP are improved by 1.58×, 2.53× and 1.72×, respectively. Part of the reason is that [14] uses an additional twiddle factor generator, which increases additional resource overhead. Another important reason is that our butterfly unit employs the fully-pipelined reconfigurable structure and avoids the separate pre/post-processing, thereby reducing the computational overhead and resources occupation.

Roy *et al.* [17] presented a custom co-processor for BFV scheme on FPGA. They applied the arithmetic optimization techniques of BEHZ [12] and HPS [13] to avoid costly multi-precision arithmetic respectively. We only list the performance and area of a single co-processor with faster architecture, which has seven residue polynomial arithmetic units (each further contains two cores), two basis extension units and two scaling units, respectively. Due to 1.52× higher frequency and 14.19× less latency on Zynq UltraScale+ FPGA, the throughput of extended ReMCA is improved by 15.13×, and the ATPs measured by LUT, FF, BRAMs and DSP achieve 14.18×, 7.37×, 10.18× and 5.17× improvements. If we run the extended ReMCA in the same frequency, the speedups of throughput and ATPs of ReMCA still reach 7.87×, 7.32×, 3.83×, 5.24× and 2.74×, respectively. Since the computational time of ReMCA excludes the overhead of intermediate data transfers during the relinearization, if we subtract this overhead (roughly 30%) from [17], benefit from the highly unified multi-channel multi-PE architecture, the throughput and ATPs of ReMCA still maintain 5.51×, 5.12×, 2.68×, 3.67× and 1.92× respectively. We also measure and report the power consumption of ReMCA on Zynq Ultrascale+ FPGA using the Vivado Power Estimator. When running at 380MHz, the result shows that ReMCA has a peak power consumption of 9.5 W. In comparison, the peak power of ReMCA is 1.24× larger due to the larger number of FFs, BRAMs and DSPs, and higher clock frequency.

Follow the work of [17], Turan *et al.* [18] further designed a high-performance coprocessor architecture with the same number of cores for RNS-BFV and implemented it on Amazon AWS FPGA. Though they employed similar optimization techniques and hardware architecture as [17], they achieved a minor performance improvement benefiting from a better hardware/software interface design. When normalized to the same frequency of [18], the throughput of ReMCA is still improved by 7.66×, and its ATPs measured by LUT, FF,

TABLE IV
PERFORMANCE COMPARISON OF HOMOMORPHIC EVALUATION OF RNS-BFV

Design	**Platform**	***N***	***logq_i* (bit)**	**No. bases**	**Freq (MHz)**	**Cycles**	**Time (μs)**			**Thr (MB/s)**	**LUT /ATP††**	**FF /ATP††**	**BRAM /ATP††**	**DSP /ATP††**
							Mult	**Relin**	**Total**					
Cathébras [14]	Virtext-7	4096	30	4	200	–	300	300	600	97.66	54188 /32.51	66444 /39.87	208 /0.12	517 /0.31
ReMCA	Virtext-7	4096	32	4	250	110592	352.3	90.1	442.4	141.27	46591 /20.61	35551 /15.73	392 /0.17	400 /0.18
Roy [17]	Zynq UltraScale+	4096	30	6	200	5349567	–	–	4458	19.72	63522 /283.18	25622 /114.22	388 /1.73	208 /0.93
Turan [18]	Virtex UltraScale+	4096	30	6	200	–	–	–	4340	20.25	57877 /251.19	33989 /147.51	305††† /1.32	208 /0.90
ReMCA	Zynq UltraScale+	4096	32	6	380	120832	255.6	58.6	314.2	298.38	64057 /20.13	49307 /15.50	552 /0.17	560 /0.18

†Throughput (Thr) = N×(No. of bits (q))/Time. ††ATP=No. LUT or FF or BRAM or DSP multiply the total time (second); †††No. BRAM = No. BRAMs (249) + No. URAMs (56).

BRAM and DSP are sped up by 6.88×, 4.95×, 4.0× and 2.65×, respectively. Similarly, if we cut the 30% data transfer overhead of [18] during the relinearization, ReMCA still achieves 5.36× throughput improvement and 4.82×, 3.47×, 2.8× and 1.86× ATPs improvements, respectively.

In terms of flexibility and scalability, since [14] adopts a streaming architecture and the computing units do not support the other functions of RNS-BFV, their architecture cannot scale with the size of evaluation circuit. As for [17] and [18], they employs a multi-coprocessor multi-core architecture, where each coprocessor is responsible for the operations in one channel and different cores support various specific functions, but the computing units in each RNS channel or between RNS channel always lack configurability even if they have many common units, which greatly reduces the resource utilization of common units. Compared with these works, ReMCA can flexibly support different primitive operations and different size circuits by configuring the structure and the number of PEs. ReMCA has high hardware flexibility and scalability, and can be widely used to accelerate FHE-based personal information sorting and retrieval, data statistical analysis, genomic testing and machine learning *et al*.

D. Comparison With Other Platforms or Schemes

Badawi *et al*. [34] presented a high performance BFV implementation on CPU and GPUs. For the parameter set ($N = 4096$, $\log(q) = 60$-bit, $\log(q_i) = 30$-bit), their single-threaded and 26-threaded software implementations require 6.33 ms and 2.84 ms respectively to compute one homomorphic multiplication on Intel (R) Xeon (R) Platinum (108 cores, 2.10 GHz). Their highly optimized GPU implementations on Tesa K80 (2496 cores, 0.82GHz) and Tesla V100 (5120 cores, 1.38GHz) require 1.98ms and 0.98 ms, respectively. We estimate that the computation times of their implementation would increase at least twice for the 128-bit modulus. As a result, compared with their CPU implementation, ReMCA achieves 14.38× and 6.45× performance improvements, respectively, while for GPU implementation, ReMCA still achieves 4.5× and 2.22× speedups, respectively.

A FPGA-based domain specific accelerator was proposed by Pöppelmann *et al*. [35] to accelerate the YASHE scheme [36], which is a very similar FHE scheme to BFV. Their implementation has the same parameter set as ours and requires 6.75 ms to compute one homomorphic multiplication when running at 100MHz. Even with faster scheme (3× to 4× faster than BFV) and smaller memory requirement (roughly half of that BFV), their implementation is still 6.14× slower than that of ReMCA under the same clock frequency.

VII. CONCLUSION

This paper focuses on accelerating the full RNS variant of BFV scheme via a well-designed reconfigurable multi-core architecture. By merging pre/post-processing into NTT/INTT and unify the read/write structure, the computational complexity is greatly reduced. The number of PEs and RNS channels of the architecture can be flexibly adjusted, and each PE is reconfigurable. A unified computing model and the mapping methods of different computing units that RNS-BFV involved are proposed. We demonstrate that our design has obvious advantages in the performance and area efficiency.

REFERENCES

[1] X. Wang, S. Yin, H. Li, L. Teng, and S. Karim, "A modified homomorphic encryption method for multiple keywords retrieval," *Int. J. Netw. Secur.*, vol. 22, no. 6, pp. 905–910, 2020.

[2] M. Ghadamyari and S. Samet, "Privacy-preserving statistical analysis of health data using Paillier homomorphic encryption and permissioned blockchain," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2019, pp. 5474–5479.

[3] A. Wood, K. Najarian, and D. Kahrobaei, "Homomorphic encryption for machine learning in medicine and bioinformatics," *ACM Comput. Surv.*, vol. 53, no. 4, pp. 1–35, Jul. 2021.

[4] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.

[5] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2009, pp. 169–178.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.

[7] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," in *Proc. IACR Cryptol. ePrint Arch.*, 2012, p. 144. [Online]. Available: https://eprint.iacr.org/2012/144

[8] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Proc. Annu. Cryptol. Conf.* Berlin, Germany: Springer, 2012, pp. 868–886.

[9] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes FV and YASHE," in *Proc. Int. Conf. Cryptol. Afr.* Marrakesh, Morocco: Springer, 2014, pp. 318–335.

[10] W. Hart, F. Johansson, and S. Pancratz, "FLINT–Fast library for number theory," 2011. [Online]. Available: http://www.flintlib.org/flint-2.0.pdf

[11] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "NFLlib: NTT-based fast lattice library," in *Proc. Cryptographers' Track RSA Conf.* San Francisco, CA, USA: Springer, 2016, pp. 341–356.

[12] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Proc. Int. Conf. Sel. Areas Cryptogr. (SAC)*. St. John's, NL, Canada: Springer, 2016, pp. 423–442.

[13] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Proc. Cryptographers' Track RSA Conf.* San Francisco, CA, UAS: Springer, 2019, pp. 83–105.
[14] 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. Cybern. Cryptograph. Hardw. Embedded Systems.*, vol. 2018, no. 3, pp. 69–88, Aug. 2018.
[15] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 353–362, Feb. 2020.
[16] 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.
[17] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 387–398.
[18] F. Turan, S. Sinha Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020.
[19] 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.
[20] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi, and A. Aysu, "A flexible and scalable NTT hardware: Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 346–351.
[21] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Trans. Comput.*, early access, Aug. 19, 2020, doi: 10.1109/TC.2020.3017930.
[22] X. Feng and S. Li, "Accelerating an FHE integer multiplier using negative wrapped convolution and ping-pong FFT," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 66, no. 1, pp. 121–125, Jan. 2019.
[23] M. Sadegh Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," 2019, *arXiv:1909.09731*.
[24] J. L. Cathebras, "Hardware acceleration for homomorphic encryption," Ph.D. dissertation, Univ. Paris-Saclay, Gif-sur-Yvette, France, 2018.
[25] G. Xin *et al.*, "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 8, pp. 2672–2684, Aug. 2020.
[26] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Instruction-set accelerated implementation of CRYSTALS-kyber," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 11, pp. 4648–4659, Nov. 2021.
[27] 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.*, vol. 27, no. 10, pp. 2459–2463, Oct. 2019.
[28] 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.
[29] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," 2019, *arXiv:1910.07557*.
[30] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT," *IACR Trans.*, vol. 2020, no. 2, pp. 49–72, 2020.
[31] A. H. Karp and P. Markstein, "High-precision division and square root," *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 561–589, Dec. 1997.
[32] 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.
[33] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. 44th Symp. Theory Comput. (STOC)*, 2012, pp. 1219–1234.
[34] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, pp. 941–956, Apr. 2021.
[35] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst. (CHES)*. Saint-Malo, France: Springer, 2015, pp. 143–163.
[36] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *Proc. IMA Int. Conf. Cryptogr. Coding*. Berlin, Germany: Springer, 2013, pp. 45–64.

Yang Su received the B.S. and M.S. degrees in engineering of micro electronics and solid-state electronics from the Information Engineering University of PLA in 2009 and 2012, respectively. He is currently pursuing the Ph.D. degree with the Rocket Force University of Engineering. He is a Lecturer with the Engineering University of People's Armed Police. His research interests include fully homomorphic encryption hardware accelerator design, reconfigurable crypto chip, and post-quantum cryptography.

Bai-Long Yang received the B.S. and M.S. degrees in computer applications technology from the Rocket Force University of Engineering, Xi'an, China, in 1990 and 1993, respectively, and the Ph.D. degree in aeronautics and astronautics manufacturing engineering from the Rocket Force University of Engineering in 2001. He is currently a Professor with the Rocket Force University of Engineering. His research interests include homomorphic encryption based on lattice, network security, and post-quantum cryptography.

Chen Yang (Member, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2004, and the M.S. and Ph.D. degrees from the Institute of Microelectronics, Tsinghua University, in 2007 and 2016, respectively. He was with VIA Technologies, Inc., Beijing, from 2007 to 2009. Currently, he is an Associate Professor with the School of Microelectronics, Xi'an Jiaotong University. His current research interests include hardware security, homomorphic encryption acceleration, reconfigurable computing, and VLSI SoC design.

Song-Yin Zhao received the B.S. and M.S. degrees in electronics science and technology from the Information Engineering University of PLA in 2014 and 2016, respectively. He is a Lecturer with the Engineering University of PAP. His research interests include artificial intelligence, hardware accelerator design, and post-quantum cryptography.