Accelerator for Computing on Encrypted Data

Sujoy Sinha Roy*, Ahmet Can Mert*, Aikata*, Sunmin Kwon †, Youngsam Shin† and Donghoon Yoo†
* IAIK, Graz University of Technology, Austria
† Samsung Advanced Institute of Technology, Suwon, Republic of Korea
*{Sujoy.Sinharoy,Ahmet.Mert,Aikata}@iaik.tugraz.at †{sunmin7.kwon,youngsam.shin,say.yoo}@samsung.com

***Abstract*—Fully homomorphic encryption enables computation on encrypted data, and hence it has a great potential in privacy-preserving outsourcing of computations. In this paper, we present a complete instruction-set processor architecture 'Medha' for accelerating the cloud-side operations of an RNS variant of the HEAAN homomorphic encryption scheme. Medha has been designed following a modular hardware design approach to attain a fast computation time for computationally expensive homomorphic operations on encrypted data. At every level of the implementation hierarchy, we explore possibilities for parallel processing. Starting from hardware-friendly parallel algorithms for the basic building blocks, we gradually build heavily parallel RNS polynomial arithmetic units. Next, many of these parallel units are interconnected elegantly so that their interconnections require the minimum number of nets, therefore making the overall architecture placement-friendly on the implementation platform. As homomorphic encryption is computation- as well as data-centric, the speed of homomorphic evaluations depends greatly on the way the data variables are handled. For Medha, we take a memory-conservative design approach and get rid of any off-chip memory access during homomorphic evaluations.**

Our instruction-set accelerator Medha is programmable and it supports all homomorphic evaluation routines of the leveled fully RNS-HEAAN scheme. For a reasonably large parameter with the polynomial ring dimension 2^{14} and ciphertext coefficient modulus 438-bit (corresponding to 128-bit security), we implemented Medha in a Xilinx Alveo U250 card. Medha achieves the fastest computation latency to date and is almost $2.4\times$ faster in latency and also somewhat smaller in area than a state-of-the-art reconfigurable hardware accelerator for the same parameter.

I. Introduction

Cloud computing services are very popular and provide high-performance computational resources to the users [2]. Despite its advantages, conventional cloud computing has security and privacy risks as the data of the user, becomes visible (as plaintext) during any computation in the cloud. Isolation techniques are followed with certain trust assumptions. Yet, in recent years several data leaks have been reported.

The concept of Fully Homomorphic Encryption (FHE) was introduced by Rivest *et al.* [24] in 1978. FHE enables logical and arithmetic operations on encrypted data without requiring any decryption of the data. Therefore FHE has a great potential in privacy-preserving outsourcing of computation to the cloud without needing to trust the cloud or a third party.

Since its inception, it took three decades to build a homomorphic encryption scheme that could perform many computations on encrypted data. In 2009, Gentry constructed the first FHE scheme [15]. FHE quickly gained interest from both academia and industry and in the last 10 years, and because of the research, better and better FHE schemes started appearing with orders of magnitude improvements in the performance.

There are several FHE or leveled FHE schemes in the literature. The difference between an FHE and a leveled-FHE is that the latter one could perform computations correctly only up to a certain complexity level whereas the first one could do arbitrary computations. It is possible to transform a leveled-FHE into an FHE by introducing a special procedure 'bootstrapping'. For evaluating arithmetic operations homomorphically, the BFV [13] and BGV [8] schemes are popular. Whereas the THFE scheme [11] is efficient for evaluating Boolean gates. On the other hand, for performing computations on encrypted *real numbers*, the HEAAN [10] and its Residue Number System (RNS) variant RNS-HEAAN [9] schemes are efficient. In fact, RNS-HEAAN is to this date the fastest scheme for performing approximate computations on the encrypted *real* data, thus making it popular for privacy-preserving machine learning applications on clouds. There are also various efforts in the literature for improving the performance of FHE using hardware accelerators [14], [23], [25], [28], [30].

Contributions: In this work, we propose a programmable instruction-set architecture (ISA), which we call 'Medha' for accelerating the Cloud-side operations of RNS-HEAAN [9]. When Medha is implemented in a single Xilinx U250 FPGA Alveo Card, it achieves around $137\times$ speedup compraed to the optimized SEAL software [27]. In greater detail, we summarize our contributions as follow:

- Implementing a high-performance processor architecture for a leveled FHE scheme, such as RNS-HEAAN, is full of challenges as homomorphic encryption is computation and data-centric. For the primitives used in RNS-HEAAN, we design hardware-friendly algorithms so that we can parallelize and pipeline them efficiently in hardware.
- Besides optimizing the number of compute cycles, a significant effort is put on the on-chip memory organization of the architecture. Although U250 has one of the largest FPGAs, reaching a high memory utilization is rather challenging as the memory elements are distributed across the FPGA floor. We follow a memory-conservative design approach and eliminate off-chip data transfers (which is slow) during homomorphic computing.
- Starting from optimized primitives, we gradually build high-level instruction-set processing elements which we call the 'RPAU'. Many RPAUs (each one is quite big) are

instantiated in parallel and then they are connected in a 'ring' structure to make the overall architecture placement and routing efficient.
- At the highest level of the implementation hierarchy, we apply instruction-level parallelism to execute a major portion of the polynomial arithmetic operations in parallel. With this, we observe around 40% reduction in the cycle count at the cost of around 20% increase in the resources.

The amount of computation that can be performed using a leveled FHE is determined by the parameter set of the implementation. Our Medha has been optimized to support the following parameter of RNS-HEAAN: the polynomial ring dimension is $N = 2^{14}$ and the ciphertext modulus is $\log_2 Q$ = 438 bits. With this parameter set, we can perform computations up to seven levels satisfying a bit security of 128 bits. Thus, Medha could be used to accelerate the cloud side homomorphic evaluations of various approximate computations such as machine learning, and neural network models, e.g., training a 2-layer CNN, logistic and exponential computation up to depth 4 [19], etc.

II. Background

In a typical homomorphic encryption protocol, there are two parties: a client and a cloud server. The cloud contains data encrypted (i.e., ciphertext) by the client, and the client performs computations on its encrypted data directly in the cloud. At the end of this protocol, the client receives the encrypted results from the cloud and performs decryptions to recover the plaintext results.

An ideal lattice-based homomorphic encryption scheme works as follows. Let, a client's secret-key be $sk = (1, s) \in R_Q^2$ and the corresponding public-key be $pk = (b, a) \in R_Q^2$. Each key is a pair of polynomials in the polynomial ring R_Q where Q is the coefficient-modulus. Client encrypts a message m using pk and obtains the ciphertext $ct \leftarrow (c_0 = r \cdot b + e_0 + m, c_1 = r \cdot a + e_1) \in R_Q^2$ where e_i is a Gaussian distributed error-polynomial. Let, a cloud contains two ciphertexts $ct = (c_0, c_1)$ and $ct' = (c'_0, c'_1) \in R_Q^2$ of the client as encryptions of messages m and m' respectively. The cloud can compute a valid encryption of $m + m'$ simply by adding the two ciphertexts as $ct_{\text{add}} \leftarrow (c_0 + c'_0, c_1 + c'_1) \in R_Q^2$. Computing an encryption of $m \cdot m'$ is relatively complex and involves several steps. First, the two ciphertexts are multiplied to obtain $ct_{\text{mult}} = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1) \in R_Q^3$. This intermediate result has three polynomial components and could be decrypted using $(1, s, s^2)$ but not using $sk = (1, s)$. Next, a special operation known as the 'Key-Switching', is used to transform the three-component ciphertext ct_{mult} (which is decryptable under $(1, s, s^2)$) into a two-component ciphertext ct_{relin} decryptable under $(1, s)$. In this context, the key-switching is called re-linearization as it obtains a linear ciphertext from a quadratic one.

The above-mentioned general framework is used in several lattice-based FHE schemes, e.g., BGV [8], BFV [13], and HEAAN [9]. Our instruction-set accelerator Medha has been designed keeping in mind the above-mentioned general framework, and as a case study, Medha has been optimized and implemented for an RNS variant of the HEAAN scheme which is popularly known as the 'RNS-HEAAN' scheme [9]. In RNS-HEAAN, the ciphertext modulus $Q = \prod_{i=0}^{L-1} q_i$ is a product of small primes q_i. These primes form the RNS basis of the implementation. With the application of RNS, a polynomial $a \in R_Q$ is represented as a vector of L residue polynomials (hence small coefficients) in the RNS basis. The biggest advantage is that these small-coefficient residue polynomials can be processed efficiently and in parallel. Hence, RNS-HEAAN is more efficient and implementation-friendly than the original HEAAN scheme [10].

Due to the page limit, we briefly describe the RNS-HEAAN scheme. To get a detailed description of RNS-HEAAN, the readers may follow the original publication [9]. To use RNS-HEAAN in an application, the first step will be to set up the scheme parameters such as polynomial-degree, modulus size, RNS-basis, etc., depending on the multiplicative depth required by the application. After that, a client generates its private-key $sk = (1, s) \in R_Q^2$, public-key $pk = (b, a) \in R_Q^2$ and a special key $\texttt{KSK} = (\texttt{KSK}_0, \texttt{KSK}_1)$ for performing the key-switching operation after a ciphertext multiplication. Each of $\texttt{KSK}_0$ and $\texttt{KSK}_1$ is a vector L of polynomials where each polynomial resides in R_{pQ} and p is a special prime modulus. After generating the keys, the client sends its public and key-switching keys to the cloud. Due to efficiency reasons, the cloud keeps these keys in the RNS representation and the NTT domain. The Number Theoretic Transform or NTT enables fast polynomial multiplications (we will see it later). Note that in the RNS representation, a polynomial in R_Q (or R_{pQ}) is a vector of L (or $L + 1$) residue polynomials. Hence, each of $\texttt{KSK}_0$ and $\texttt{KSK}_1$ has $L \cdot (L+1)$ residue polynomials. The Client-side operations are relatively a lot simpler than the Cloud-side operations. Our Medha accelerates the Cloud-side operations.

RNS-HEAAN subroutines used in the Cloud: In the following part, we use the notation Q_l to represent the ciphertext-modulus at level l and $Q_l = \prod_{i=0}^{l-1} q_i$ with $l \leq L$. It implicitly performs all arithmetic operations on the residue polynomials.

- $\texttt{HE.Add}(ct, ct')$: It adds the respective polynomials of the two ciphertexts and outputs the result.
- $\texttt{HE.Mult}(ct, ct')$: It multiplies two input ciphertexts $ct = (c_0, c_1) \in R_{Q_l}^2$ and $ct' = (c'_0, c'_1) \in R_{Q_l}^2$, and computes $d_0 = c_0 \cdot c'_0 \in R_{Q_l}$, $d_1 = c_0 \cdot c'_1 + c_1 \cdot c'_0 \in R_{Q_l}$, and $d_2 = c_1 \cdot c'_1 \in R_{Q_l}$. The output is the non-linear ciphertext $d = (d_0, d_1, d_2) \in R_{Q_l}^3$.
- $\texttt{HE.Relin}(d, \texttt{KSK})$: It re-linearizes the result of previous step and produces a ciphertext that is decryptable under the secret key. Let $d_{2,i} = d_2 \pmod{q_i}$ for $0 \leq i < l$. Now compute $ct'' = (c''_0, c''_1)$ where $c''_0 = \sum_{i=0}^{l-1} d_{2,i} \cdot \texttt{KSK}_0[i] \in R_{pQ_l}$ and $c''_1 = \sum_{i=0}^{l-1} d_{2,i} \cdot \texttt{KSK}_1[i] \in R_{pQ_l}$. Finally, output the re-linearized ciphertext $ct_{\text{relin}} = (d_0, d_1) + \lfloor p^{-1} \cdot ct'' \rceil \pmod{Q_l}$.

Fig. 1 shows the hierarchy of different operations that are used in a homomorphic application. At the highest level of this hierarchy, there are homomorphic procedures for performing

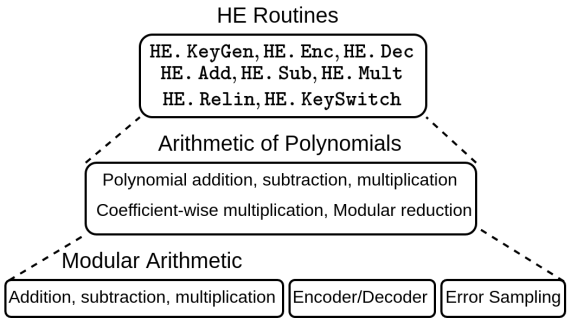

Fig. 1. Implementation hierarchy of homomorphic encryption. Key generation, encryption, and decryption are performed on the user side. These user-side operations also include encoding, decoding, and Gaussian sampling at the lowest level. Homomorphic evaluations on ciphertexts using HE. Add, subtract, multiply, key-switching, etc., are performed at the cloud side. Our hardware accelerator is designed for accelerating cloud-side operations which are significantly more expensive than user-side operations.

TABLE I
PARAMETER SETS

Parameter Set	$(N, \log_2 Q)$	$L+1$	Mult. Depth	Security Level[1]
Set-1	$(2^{14}, 384)$	7	6	151-bit
Set-2	$(2^{14}, 438)$	8	7	130-bit
Set-3	$(2^{14}, 492)$	9	8	115-bit

[1]: Results are obtained using lwe security estimator [1].

computations (e.g., addition, multiplication, key-switching, etc.) on the ciphertexts. These high-level operations translate into the arithmetic of polynomials: polynomial addition, polynomial subtraction, polynomial multiplication, coefficient-wise multiplication, coefficient-wise modular reduction, and coefficient-wise scalar multiplication. Finally, the lowest level of this hierarchy is composed of arithmetic blocks for residue arithmetic where the moduli are 54 or 60-bit primes.

Parameter set for our implementation For proof-of-concept, we implement three different versions of the accelerator with three-parameter sets, which we refer to as Set-1, Set-2, and Set-3. They have the coefficient modulus sizes 384, 438, and 492, respectively, with the same polynomial degree ($N = 2^{14}$) as shown in Table I.

III. COMPUTATION AND PLATFORM CONSTRAINTS

A. Profiling for identifying computational challenges

Before designing a hardware accelerator for FHE, it is important to know which of the above-mentioned computation primitives are the most expensive ones, so that we can allocate more resources for them in the architecture. Software profiling of the SEAL [27] library on an Intel i5 laptop showed that polynomial multiplications are the most computation-intensive operations. The polynomial multiplications in SEAL use the Number Theoretic Transform (NTT) method that gives the fastest asymptotic time complexity of $O(n \log n)$. The NTT and inverse NTT (INTT) computations take roughly 67% and 7% of the overall computation time of a key-switching operation when the polynomials are 2^{14} coefficients, and there are eight moduli in the RNS base. The remaining 26% is spent on different coefficient-wise operations.

Handling huge active data variables is very important for performance. For the above-mentioned parameter, to perform a homomorphic multiplication, SEAL allocates around 600 MBs of memory in the stack and heap. The total on-chip memory (BRAMs and URAMs) in the target FPGA is only around 24 MBs. Additionally, it is generally not feasible to achieve 100% utilization of the on-chip memory as the BRAMs and URAMs are distributed throughout the FPGA.

B. Overview of Alveo U250 Platform

We have implemented Medha in the Alveo U250 data center accelerator card. It contains one of the largest Xilinx FPGAs. The FPGA is divided into four isolated 'Super Logic Regions (SLRs)'. Such a splitting leads to lower power and energy consumption. An SLR is connected to its neighboring SLR(s) through stacked silicon interconnects. These inter-SLR connections are limited in number. Hence, for implementing a large architecture on a multi-SLR FPGA, the number of nets crossing the SLRs should be sufficiently small.

All the SLR regions have an almost similar amount of logic and memory resources. In our case, each SLR has around 345K LUTs, 705K FFs, 2877 DSPs, 500 BRAMs, and 320 URAMs. Integer multipliers are generally implemented using DSPs instead of LUTs for better speed and area. The BRAMs and URAMs are addressable dual-port memory elements. URAMs are profitable when we have to store big but less frequently-accessed data in the on-chip memory. The BRAMs and URAMs are arranged in columns in the SLRs.

IV. IMPLEMENTATION OF MODULAR ARITHMETIC

The ground layer of the implementation hierarchy in Fig. 1 is built of operators for the arithmetic of residue numbers. The prime moduli are only 54 to 60 bits large. We use hardware-friendly algorithms and optimizations so that we can optimize these fundamental arithmetic blocks for both area and speed. Modular addition and subtraction are the simplest blocks and they are implemented using adder and subtractor circuits made of configurable fabric logic, i.e., LUTs.

For implementing *modular multiplication*, we instantiate bit-parallel modular multipliers so that the highest throughput can be achieved. A bit-parallel modular multiplier is essentially an integer multiplier followed by a reduction circuit. Integer multipliers are implemented using DSP slices with several layers of pipeline registers to meet at least 300 MHz clock-frequency constraints. One 60-bit integer multiplier consumes 10 DSP slices.

The most commonly used methods for performing modular reduction are based on the Barrett or Montgomery methods. The SEAL library [27] uses the Barrett reduction technique for reducing the results of integer multiplications. The same technique is also used in the hardware architecture of the HEAX processor. In the Barrett reduction, the input (i.e., the non-reduced value) is first multiplied by a constant, and then a bit shifting is performed to obtain a quotient. Next, a partially reduced result is calculated by multiplying the quotient by

another constant. The final result requires a conditional subtraction operation. In summary, the Barrett reduction method is based on multiplications. Another method for implementing the modular reduction operation is using a table-based modular reduction approach as proposed in [25]. In this method, the result of a multiplication is reduced in multiple steps where each step reduces a part near the most significant bit using a pre-computed look-up table. Although this method does not use any DSP multipliers, it increases LUT utilization on FPGA and does not provide an optimal solution when the modulus is constant. When the modulus is selected as a sparse prime, the reduction operation can be performed efficiently using only add and shift operations as proposed in [34]. In this work, we use reduction-friendly primes as moduli and employ the fast add-shift-based modular reduction method to save both DSPs and LUTs [34]. For example, the first modulus in the RNS-basis is a 60-bit sparse prime $q_0 = 2^{59} + 2^{20} - 2^{15} + 1$ and the result of a multiplication (120 bit) is reduced using the relation $2^{59} \equiv -2^{20} + 2^{15} - 1 \pmod{q_0}$ recursively. This approach saves up to 45% LUT units compared to the table-based modular reduction method.

V. Implementation of Polynomial Arithmetic

In the second level of the implementation hierarchy in Fig. 1, large-degree polynomials are processed. Other than the polynomial multiplication, the remaining operations are coefficient-wise with $O(n)$ time complexity.

For *multiplying two large polynomials*, the Number Theoretic Transform (NTT) that is a generalization of the Fast Fourier Transform (FFT), is the fastest method thanks to its $O(n \log n)$ time complexity. To perform a multiplication, the input polynomials are transformed into frequency-domain polynomials, then they are multiplied coefficient-wise, and finally, the result is brought to the time-domain by applying the inverse NTT (INTT).

We use the decimation-in-time (DiT) approach for the NTT and decimation-in-frequency (DiF) approach for the INTT [26]. This particular combination eliminates the need for any coefficient permutations before the NTTs or after the INTT as used in SEAL [27]. The DiT NTT algorithms is described in Alg. 1. The butterfly operation in Alg. 1 takes a pair of coefficients, then performs modular arithmetic (addition, subtraction and multiplication), and finally produce two new coefficients. To remain within the page limit, we do not describe the DiF INTT algorithm. In comparison to the DiT method, the DiF method processes the polynomial coefficients in the reverse order and the butterfly computations are performed differently.

A. Parallel NTT architecture

On hardware platforms, the NTT algorithm can be parallelized by instantiating parallel compute cores for the butterfly operation. At the same time, due to the loop-dependent varying memory access pattern (Alg. 1), implementing a highly parallel NTT [23], [25], [28] requires solving three major challenges: memory-access bottleneck due to port limitation, memory read/write conflicts, and long routing nets

Algorithm 1 Decimation-in-time (DIT) forward NTT

Input: Polynomial $a(x) \in R_q$ and $\omega_N \in \mathbb{Z}_q$ (N-th root of unity)
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}]$ = NTT(a)

1: $A \leftarrow$ BitReverse(a) ▷ permutation of coefficients
2: **for** $m = 2$ to N by $m = 2m$ **do**
3: $\omega_m \leftarrow \omega_N^{N/m}$
4: $\omega \leftarrow 1$
5: **for** $j = 0$ to $m/2 - 1$ **do** ▷ butterfly loop
6: **for** $k = 0$ to $n - 1$ by m **do**
7: $(u, t) \leftarrow (A[k + j], A[k + j + m/2])$
8: $A[k + j] \leftarrow u + \omega \cdot t$
9: $A[k + j + m/2] \leftarrow u - \omega \cdot t$
10: **end for**
11: $\omega \leftarrow \omega \cdot \omega_m$
12: **end for**
13: **end for**

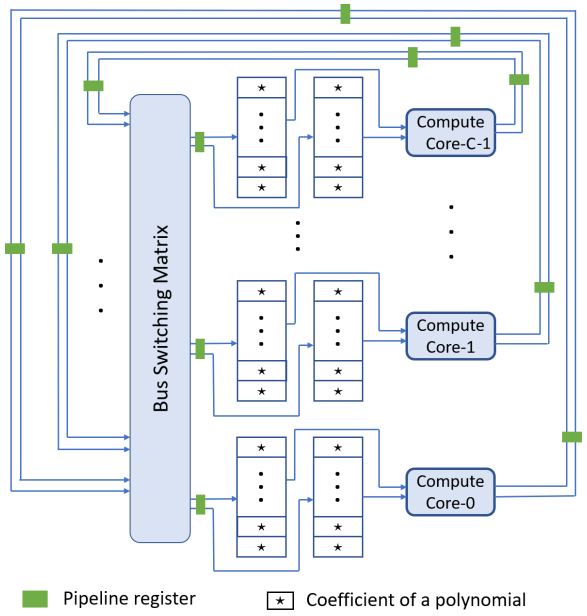

Fig. 2. Organization of memory and compute cores for efficient implementation of parallel NTT

between compute-cores and memory elements. Taking inspiration from [23], [25], [28] we implement our parallel NTT unit. Fig. 2 shows a high-level organization of the memory elements (for storing the polynomial) and the compute cores (for processing the coefficients). As the on-chip memory elements (e.g., BRAM) in FPGAs are dual-port, one of the ports is used for reading, and the other port is used for writing during an NTT. Each compute core is responsible for computing one butterfly operation every cycle.

Following [25] any compute core in Fig. 2 reads its operand data exclusively from a fixed set of memory elements. With this strategy *'all memory element'* -to- *'all compute core'* communication which requires $O(c^2)$ nets (where c is the number of cores) is reduced to $O(c)$ nets only.

Output coefficients from the compute cores are stored in the memory through the 'bus switching matrix' in Fig. 2. The matrix rearranges the received coefficients in such a way that during the next loop-iteration of the NTT algorithm, each compute core gets its operand coefficients from its exclusive memory elements. The matrix is composed of a set

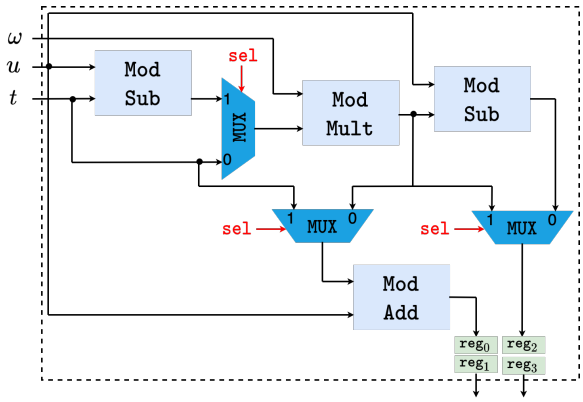

Fig. 3. Organization of the unified butterfly core (without pipeline registers) for DiT and DiF NTTs. During DiT, the selection signal *sel*=0 for computing $\{u + t \cdot \omega, u - t \cdot \omega\}$. During DiF, *sel*=1 to compute $\{u + t, (u - t) \cdot \omega\}$.

of multiplexers and its area is proportional to the logarithm of the number of compute cores.

The data paths that originate at the outputs of the compute cores and reach the bus switching matrix (Fig. 2) are long. We put multiple layers of pipeline registers (shown in green in the figure) to split these long paths into much shorter paths. This pipeline strategy increases the clock frequency without increasing the latency as each compute core produces the result-coefficients in a continuous 'stream'.

The number of parallel compute cores is a design parameter that depends on area and performance budgets. Our NTT unit uses $c = 16$ butterfly cores and computes one NTT in around 7,168 cycles for polynomials of $N = 2^{14}$ coefficients. Each memory element in Fig. 2 is a dual-port BRAM36k slice and hence a butterfly core exclusively reads a pair of BRAM36k to obtain two coefficients every cycle.

B. Unified butterfly core for DiT NTT and DiF INTT

The forward and inverse transform use different butterfly steps. In DiT a new coefficient-pair (u', t') is computed from (u, t) as follows: $(u', t') \leftarrow (u+t\cdot\omega, u-t\cdot\omega)$. Whereas in DiF a new coefficient-pair is computed as $(u'', t'') \leftarrow (u + t, (u - t)\cdot\omega)$. We use a unified butterfly core for computing both DiT and DiF methods. It is shown in Fig. 3. The multiplexers are used to select an appropriate data path depending on the DiT or DiF mode of operation.

Pipeline registers (not shown in the figure) are used to meet over 300 MHz clock frequency. We propose an optimized pipeline scheduling to reduce the area overhead of our pipelined butterfly core significantly compared to previous works [23], [25], [28]. The modular multiplier in Fig. 3 uses many more levels of pipeline registers compared to the modular adder and subtracter. Let us consider the processing of coefficient pairs during DiT NTT. After reading the coefficient t, computation of the intermediate data $\omega \cdot t$ progresses through a long chain of pipeline registers present inside the modular multiplier. For the correct computation of $(u', t') \leftarrow (u+t\cdot\omega, u-t\cdot\omega)$, u and $\omega \cdot t$ must reach the inputs of the adder and subtractor synchronously in the same cycle. In [25] both u and t are read together from the memory and then u is passed through a long chain of redundant registers

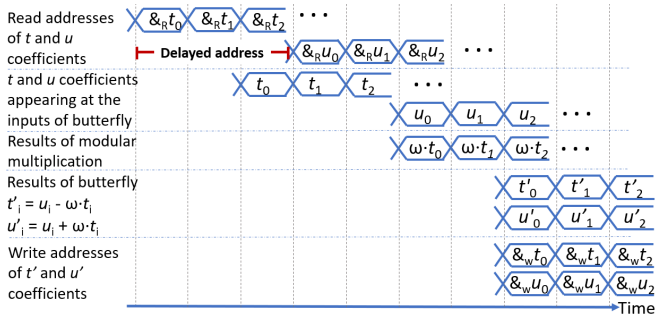

Fig. 4. The timing diagram for our DiT method of NTT. Due to pipelined datapath, the reading of the u coefficients is delayed so that we can add or subtract them when the corresponding modular multiplication results $t \cdot \omega$ are ready. The results of butterfly operations are written synchronously. For the DiF method, the read-write happen oppositely: we read the u and t coefficients synchronously but write them asynchronously. The notations $\&_R$ and $\&_W$ are for reading and writing addresses respectively.

just to make sure that both u and $\omega \cdot t$ arrive together at the adder and subtracter. We do not use this pipeline strategy as it adds an overhead of around 1,200 flip-flops per butterfly core and around 153,600 flip-flops for Medha.

Our pipeline strategy avoids the above-mentioned bloated register consumption by simply delaying the read of coefficient u from the memory. We keep u and t in separate memory elements in Fig. 2 so that they can be read separately just-in-time. The timing diagram in Fig. 4 shows how the $\{u, t\}$ coefficients are read during a DiT NTT. Reading of the t-coefficients for the consecutive butterfly operations is initiated several cycles ahead of reading the u-coefficients. The timing gap between the two reading streams equals the number of pipeline stages in the modular multiplier. As a consequence, each modular multiplication result and the respective u appear synchronously at the inputs of the adder and subtractor circuits in Fig. 3. In this way, just by delaying the read of the u-set of coefficients, we eliminate the need for any redundant registers.

We extend the above-mentioned pipeline strategy to the DiF method of INTT. The difference is that both u and t are read simultaneously but the result coefficients are written separately into the memory.

C. Twiddle factors during NTT

Software implementations (e.g., SEAL [27]) and also the hardware implementations [23], [28] save cycle count by keeping all the required twiddle constants in large tables. As several previous hardware implementations [23], [28] indicated that homomorphic encryption is memory-bound in Medha we compute the twiddle factors on the fly and in parallel to the butterfly operations so that we save on-chip memory without increasing the latency.

A twiddle factor generation unit mainly consists of a modular multiplier and only one memory element for keeping a few initial constants. The multiplier is not an additional cost as it is reused to speed-up non-NTT operations, such as coefficient-wise multiplication of two polynomials and coefficient-wise modular reductions of an input polynomial.

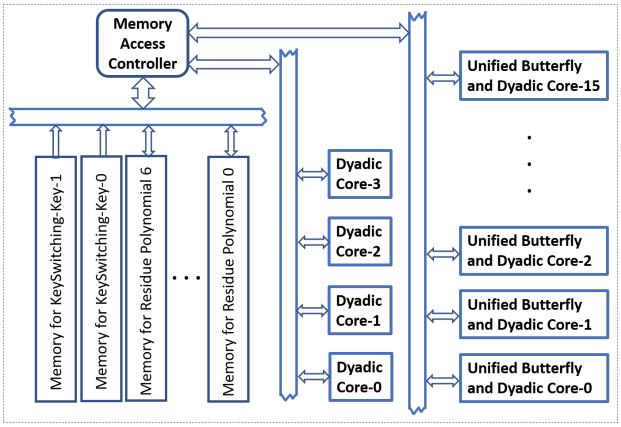

Fig. 5. Architecture of the Residue Polynomial Arithmetic Unit (RPAU).

VI. Architecture of the Homomorphic Encryption Processor

In this section, we describe the level in Fig. 1 where homomorphic computations namely, homomorphic addition/subtraction, multiplication, and key-switching are performed. Our Medha is an instruction-set architecture (ISA) and hence programmable. By changing the program-microcode, Medha could be re-used for executing different homomorphic routines. ISAs have been designed for accelerating homomorphic encryption in [25], [28]. However, their performance advantages have remained limited, e.g., only 13× speedup compared to the SW-only implementation in [28]. Medha overcomes the speed limits of the previous ISA-based implementations and offers about 137× speedup compared to the software implementations. Indeed, Medha has a faster latency than HEAX [23] which is a block-unrolled architecture.

A. High-level architecture of Medha

In an RNS homomorphic encryption scheme, an arithmetic operator is applied to a 'vector' of residue polynomials. This has some similarities with the Single Instruction Multiple Data (SIMD) processors. Our Medha has been designed to leverage that algorithmic parallelism.

At a high level, Medha instantiates parallel computation units similar to [28] for processing the residue polynomials in SIMD manner in parallel. We call one such computation unit the 'Residue Polynomial Arithmetic Unit (RPAU)'. We instantiate one RPAU per moduli of the RNS basis. Next, the RPAUs are connected carefully as data exchanges must happen between them during the key-switching or relinearization.

We would like to mention that, although similar SIMD processors [28], [30] have been proposed in the literature, we make significantly novel contributions in the internal design of the RPAUs, on-chip memory organization, and the way these RPAUs are interconnected to realize Medha. Because of all these optimizations, our Medha achieves 2-order speedup; whereas the previous ISAs [28], [30] achieved only 1-order speedup compared to a slower software and also with a 4× smaller parameter (thus less challenging to implement).

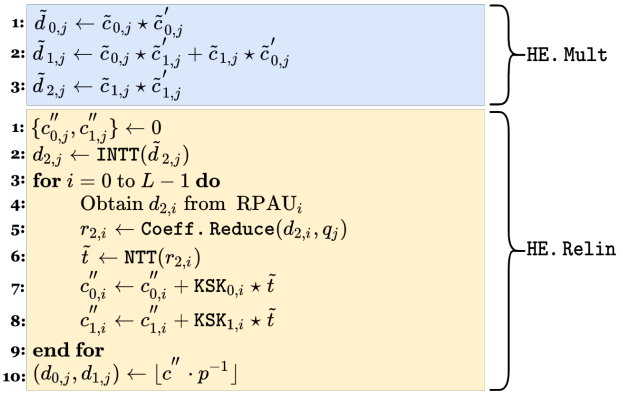

Fig. 6. Computation steps that are performed in the j-th RPAU during a homomorphic multiplication followed by a key-switching operation. The tilde is used to indicate that a data variable is in the NTT domain. Coefficient-wise multiplication of two polynomials is denoted using $\star$.

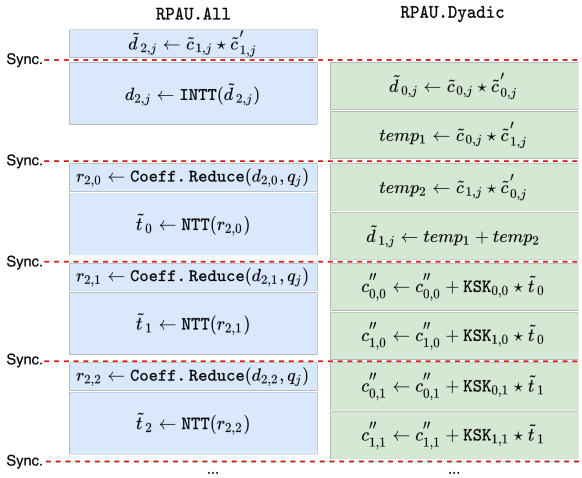

Fig. 7. Parallel processing of Fig. 6 using two threads inside an RPAU.

B. Design of Residue Polynomial Arithmetic Unit (RPAU)

An RPAU, shown in Fig. 5, is composed of polynomial arithmetic blocks and memory for storing the operand and resultant residue polynomials. There are two groups of compute cores, namely `RPAU.All` and `RPAU.Dyadic`, for executing two residue polynomial arithmetic instructions in parallel. `RPAU.All` is capable of performing all kinds of polynomial arithmetic operations that an RPAU is needed to perform. Hence, it is a necessary and sufficient component for designing an RPAU. `RPAU.Dyadic` enhances the performance of an RPAU by computing coefficient-wise (i.e., dyadic) operations in parallel to the operations performed by `RPAU.All`.

Instruction parallelism within RPAU: Looking at the data dependencies between the steps in Fig. 6, we see that some of the steps can be performed in parallel. For example, in the first block, $d_{0,j}$, $d_{1,j}$, and $d_{2,j}$ can be computed in parallel to each other as they depend only on the input polynomials. Inside the loop of the key-switching operation, the steps are sequential in nature due to data dependencies. As the loop iterates several times, we can parallelize the loop by unrolling it and then 'block-pipelining' the loop-internals. In summary, we have different options for applying parallel processing. Next, we

discuss the architectural options and the design decisions that we take to leverage this parallelism without increasing the area significantly.

- Option 1: Running more than one NTTs in parallel inside an RPAU will become essential if we unroll the loop of the key-switching operation in Fig. 6. The NTT unit that we use has 16 compute cores and they are not really small in area. Hence replicating the NTT unit multiple times will increase the area significantly.
- Option 2: Using only one NTT unit with more compute cores. Compared to the previous option, this option will be simpler as well as more effective in reducing the latency irrespective of data dependencies. E.g., instead of using 16 cores in the NTT, if we use 32 or 64 cores then we can reduce the cycle count of an NTT by a factor of 2 or 4 respectively. A potential problem is that we may not see a similar reduction in the overall computation *time* due to a slow-down in the clock frequency of the much larger architecture. Another problem is that the number of cores in NTT increases by powers of two, leaving no room for a middle solution.
- Option 3: Following Option 2, if we use only one NTT unit per RPAU, then a way to reduce latency will be to execute coefficient-wise polynomial operations in parallel with NTTs. In this approach, a few additional arithmetic cores for these cheaper coefficient-wise operations will be needed in the RPAU. For example, using only four coefficient-wise arithmetic cores, we can compute two dyadic polynomial arithmetic instructions (taking $2 \times 4,096$ cycles) concurrently with an NTT (taking 7,168 cycles). In Fig. 7 we provide a timing diagram and show how we can speedup the computation of Fig. 6 using two parallel threads.

C. Organization of on-chip memory inside RPAU

As RNS-HEAAN involves computations on large-degree residue polynomials, storage and access of the operand and intermediate results play a critical role in the performance. Off-chip data transfer is very expensive, and therefore while designing the RPAU architecture, we aimed to store all or most of the required data in the on-chip memory elements such as BRAMs and URAMs. After analyzing and optimizing the steps of homomorphic multiplication + key-switching, we observe that the peak memory requirement is equal to storing seven residue polynomials (operands and intermediate data) in each RPAU. Additionally, if we also keep the key-switching key inside the RPAU to completely avoid off-chip memory access, then we need storage for $2L$ additional residue polynomials. This is because the key-switching loop in Fig. 6 requires L polynomials from each of $\texttt{KSK}_0$ and $\texttt{KSK}_1$. For the parameter set that we are using, $L = 7$. Thus, in total, we need to store 21 residue polynomials to eliminate the need for off-chip memory access and achieve minimum computation time.

On the target FPGA platform, a URAM is 8 times larger than one BRAM36k. However, both types of memory have only two ports. Storing one residue polynomial would cost 32 BRAM36k slices or 4 URAM slices. However, we also need a high data rate for feeding the computation cores and storing results by them. For example, if a polynomial is stored using 4 URAMs, then due to the limitation in the number of ports, we cannot use all 16 cores of the NTT unit. Also in Sec. V-A we will see that for efficient placement and simpler memory access, the memory elements are required to be exclusive to the compute cores. Hence, designing the memory bank of the RPAU requires careful considerations of the computation and architectural constraints.

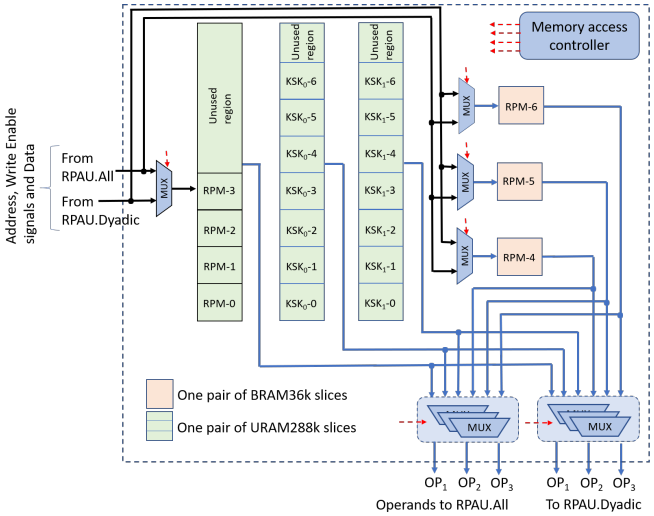

Fig. 8. Organization of memory elements inside the memory module. One such memory module is connected to a single compute core of the NTT unit. There are 16 such memory modules inside each RPAU. Any residue polynomial is split into 16 fragments, and one such fragment is stored in one RPM-i of a memory module.

We make the memory organization modular within the RPAUs. First, we align BRAMs and URAMs to implement the 'memory-bank' that will be exclusively read by one of the 16 compute cores during the NTT. In this way, the memory-bank of the i-th compute core keeps only the i-th fragments of all the 21 residue polynomials. This memory-bank is a hybrid of BRAMs and URAMs and its internal architecture is shown in Fig. 8. The abbreviation 'RPM' stands for residue polynomial memory. There are seven RPMs inside the core memory and each RPM stores 1/16-th of the consecutive coefficients, i.e., 1,024 coefficients for $N = 2^{14}$. In the figure, we use the peach and light green colors to represent RPMs that are based on BRAMs and URAMs respectively. RPM-4, 5, and 6 are composed of BRAM36k slices and are physically separated. Hence, they can be read/written in parallel. Whereas, RPM-0-to-3 are implemented using a single pair of URAMs and are logically separated. Hence, only one of them can be read and only one of them can be written every cycle. It is the job of the programmer to decide which polynomial goes to which RPM taking data dependencies and access patterns of a subroutine into consideration.

The 'Memory access controller' block is responsible for handling memory accesses of the two parallel computing threads namely, RPAU. All and RPAU.Dyadic (from the previ-

ous subsection). The two threads must use mutually exclusive RPMs at any cycle. Again, it is the job of the programmer to correctly specify the operand RPMs of the two parallel threads so that no access conflicts arise at any point in time. Based on the operand RPMs of the two threads, the memory access controller generates appropriate control signals for the multiplexers in Fig. 8.

D. Interconnecting RPAUs

The proposed processor architecture supports a large parameter set and it works with many coefficient moduli, namely $L+1$ including the special prime. So, it instantiates $L+1$ RPAUs in parallel. As the amount of logic and memory elements in each SLR of the FPGA is limited, the RPAUs are carefully placed. To that end, for a high-performance implementation, it is essential to perform the placement of building blocks effectively across multiple SLR regions to minimize the cost of routing.

Each RPAU unit uses 96 URAMs. Since each SLR region has around 320 URAMs, only 3 RPAUs can be mapped per SLR at most. For an architecture supporting $L+1=7$, we have to employ 7 RPAU units and this dictates the placement of RPAUs into at least three different SLR regions. The communication between the FPGA and host resides in SLR1, thus the input polynomials (i.e. ciphertexts) should be efficiently sent to and stored in memory blocks of RPAUs mapped to other SLR regions. Similarly, output polynomials should be read from different SLR regions to SLR1. This requires careful floorplanning of RPAUs and an efficient approach for sending data from a single location to multiple locations in a large FPGA. The naive solution would be directly connecting inputs in SLR1 to memory blocks in other SLRs. This would require separate paths for each connection and create a 'star-like' architecture, which complicates the routing, increases the number of nets crossing SLRs, and reduces the maximum achievable clock frequency significantly.

In the proposed design, each RPAU unit has access to only its local memory blocks for reducing the interconnects between RPAUs and simplifying the routing. Due to algorithmic requirements as shown in Fig. 6, there is still a need for RPAU-to-RPAU communication. The first approach for realizing this requirement was using a central memory block accessible by every RPAU unit. Then, RPAU units can use the central memory for exchanging data with each other. However, this approach would create new routing paths. Instead, we proposed a 'ring-like' architecture to reduce routing. This way, each RPAU can send its data to any other RPAU through a chain of RPAU units. For example, when RPAU#6 needs to send its data to RPAU#1, it first forwards data to RPAU#0, and then it is forwarded to RPAU#1. Thus, each RPAU will have direct connections with only two RPAUs. Since each SLR region can have at most 3 RPAUs, there have to be at least two RPAU-to-RPAU data connections between each SLR region. To keep the crossing nets between SLRs at a minimum, we tried to place consecutive RPAUs in the same SLR region. For example, RPAU#1, RPAU#2, and RPAU#3 are placed in the same SLR with a 'ring-like' architecture as shown in Fig. 9, where the 'ring' is marked with a red line.

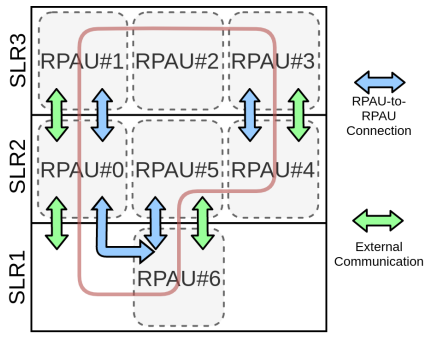

Fig. 9. The proposed 'ring' structured floorplan to minimize routing cost for the implementation with 7 RPAU units.

For external communication signals, we followed the same approach and adopted the same 'ring-like' architecture for sending data signals from SLR1 to other SLR regions as shown in Fig. 9. External communication signals are forwarded to an RPAU through a chain of RPAUs, which reduces the routing cost significantly. For example, the external communication signals first go from SLR1 to the RPAU#0 in SLR2 through three-stage of buffers. Similarly, these signals in SLR2 are forwarded to the RPAU#1 in SLR3 and so on.

E. Program Execution Unit

Our Medha is an instruction set architecture with its own program execution unit. An RPAU receives its instructions from a program execution unit. Using dedicated program controllers for each RPAU we can run asynchronously when there are no data dependencies between the RPAUs. However, the key-switching operation requires periodic data exchanges between RPAUs. Hence, we do not allocate dedicated program controllers for any RPAUs. By analyzing the computation steps in the homomorphic subroutines, we observe that most of the time the RPAUs could execute the same instruction in a SIMD manner. Only during the rescaling operation, the program execution flow splits into two parallel branches: a subset of the RPAUs follow the first branch and the remaining RPAUs follow the second branch. Hence, Medha uses only two program controllers inside its program execution unit. We would also like to mention that by reducing the number of program controllers to two from 'one for each RPAU' we greatly simplify the programming model of Medha.

F. Hardware-Software Interfacing of the Overall System

We implemented a proof-of-concept software stack (Fig. 10) consisting of a SEAL library, User-Mode Driver (UMD), and Kernel Mode Driver (KMD). The UMD provides an interface layer for SEAL, and KMD supports the scheduling of jobs. When a SEAL command (supported by Medha) is executed, the corresponding UMD-API is called to submit the command with the required parameters to KMD's job queue as a job. Next, KMD's job scheduler sends the job to Medha. When Medha completes its task, the result is read through the PCIe interface. All data communications are performed using XDMA [32] for fast transfers. We use the

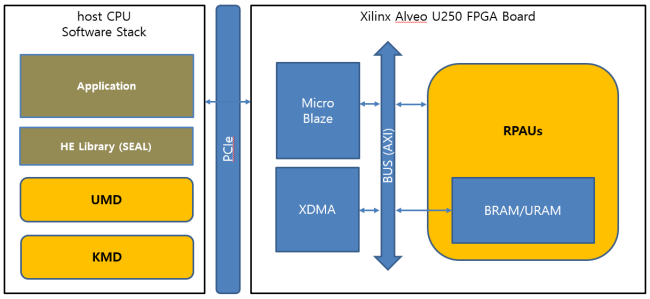

Fig. 10. CPU-FPGA interface and software stack

TABLE II
PERFORMANCE OF EACH INSTRUCTION/OPERATION

	Instruction/Operation	Clock Cycles	Lat. (μ sec)	Throug. (per sec)
Instructions	N-pt NTT	$\approx$ 7,200	36	27,777
	N-pt INTT	$\approx$ 7,200	36	27,777
	RPAU-to-RPAU broadcast	$\approx$ 512	2.56	390,625
	Scale by $q_i^{-1} \pmod{q_j}$	$\approx$ 512	2.56	390,625
	C.wise Add/Sub/Mult (main)	$\approx$ 512	2.56	390,625
	C.wise Add/Sub/Mult/MAC (dyd)	$\approx$ 4,096	20.48	48,828
Set-1	Hom. Add [1]	1,134	4.54	220,264
	Hom. Mult. + Relin. [1]	88,411	353.64	2,827
Set-2	Hom. Add	1,134	5.67	176,366
	Hom. Mult. + Relin.	96,740	483.70	2,067
Set-3	Hom. Add	1,134	5.67	176,366
	Hom. Mult. + Relin.	105,069	525.35	1,903

[1]: The implementation with 250MHz.

MicroBlaze (Xilinx's microprocessor) unit for controlling the communication between the host CPU and the RPAUs, and also for monitoring the entire FPGA system.

VII. RESULTS

The proposed architecture is described in Verilog HDL and implemented for Xilinx Alveo U250 card using Vivado 2019.1 tool with a performance-optimized implementation strategy. The implementations with the Set-1, Set-2, and Set-3 from Table I employ 7, 8, and 9 RPAUs respectively.

A. Timing results

The proposed implementation with Set-1 runs at 250MHz, and the implementations with Set-2 and Set-3 run at 200MHz. In Table II, we present the cycle count, latency (in μ sec) and throughput results for each low-level instruction and high-level operations for all three designs. Since each design employs RPAUs as the number of ciphertext coefficient modulus (i.e. the design with Set-1 has 7 RPAUs), the low-level instructions have the same clock cycles for each implementation. The low-level instructions for synchronizing main/dyadic cores, synchronizing the program controllers, and ending the program do not consume any clock cycles, thus they are not included in Table II. Since we use only on-chip memory (i.e. registers and BRAMs/URAMs) during the computations, the proposed architectures do not have any DDR data transfer overhead.

Computational cost of homomorphic multiplication: Homomorphic multiplication and relinearization operations consist of two parts as shown in Fig. 6: `HE.Mult` and `HE.Relin`. The `HE.Mult` operation is performed using coefficient-wise multiplication and addition instructions. Since the ciphertext element $\tilde{d}_{2,j}$ is required by the `HE.Relin` operation, it is first computed using the fast `RPAU.All` cores while other ciphertexts ($\tilde{d}_{0,j}$ and $\tilde{d}_{1,j}$) are computed using the `RPAU.Dyadic` cores. The `HE.Relin` operation is the most computationally intensive part, and it requires the utilization of both main and dyadic cores. The `HE.Relin` operation requires each $\tilde{d}_{2,j}$ polynomial in base q_j to be reduced by other bases. Thus, it first requires all $\tilde{d}_{2,j}$ polynomials to be converted to the polynomial domain using one INTT instruction. Then, the polynomial $d_{2,j}$ is sent to other RPAUs for reduction operation and converted back to the NTT domain. This procedure is repeated in the main core for $j = 0, 1, \ldots, L-1$ as shown in Fig. 6. At the same time, multiplication and accumulation of the resulting $\texttt{NTT}(d_{2,i})$ polynomials with key-switching keys are performed in the dyadic core as visualized in Fig. 7. This parallelism improves the performance significantly. The routine for homomorphic multiplication and relinearization operations are the same for each design except for the number of required multiplications during the `HE.Relin`. The design with Set-1, Set-2, and Set-3 require 6, 7, and 8 multiplications during the `HE.Relin`, respectively. Hence, a total of $\approx$59,904, $\approx$68,096 and $\approx$76,288 clock cycles are required during the `HE.Mult` and the `HE.Relin` operations for the designs with Set-1, Set-2, and Set-3, respectively, excluding extra cycles generated by pipeline bubbles between instructions.

The final step of `HE.Relin` operation requires the polynomials in the last coefficient base (p) to be reduced by other coefficient bases, scaled, and added with ciphertexts. Since the polynomial in base p resides in the NTT domain after the key-switching loop, it first needs to be converted to the polynomial domain with INTT operation. Then, it is reduced by other bases and converted to the NTT domain. Finally, the reduced polynomials are subtracted, scaled by $p^{-1} \pmod{q_i}$ for $i = 0, 1, \ldots, L-1$ and added with the ciphertexts $(d_{0,j}, d_{1,j})$. In the proposed architectures, we use two program controllers. This allows us to perform two different instructions for RPAUs at the same time as detailed in Section VI. This feature of the architecture enables us to perform NTT instruction for the first element of ciphertext with the first $L-1$ RPAU and perform INTT instruction for the second element of ciphertext with the last RPAU. This saves us one INTT instruction during the last step of `HE.Relin` computations. Hence, a total of $\approx$26,208 clock cycles are required for the last step of `HE.Relin` operation excluding extra cycles generated by pipeline bubbles between instructions.

B. Resource Utilization results

Detailed resource utilization of main arithmetic units for the design with Set-1 is shown in Table III. For the RPAU unit, only the utilization result of the first RPAU is presented. Other RPAU units have similar resource utilization. The 'Platform' unit which is responsible for the communication between the FPGA and the host CPU significantly contributes the resource utilization. We also provide resource utilization reports for all three designs in Table IV. Except for the URAM utilization,

TABLE III
RESOURCE UTILIZATION OF ARITHMETIC MODULES FOR SET-1

Modules	LUTs	REGs	BRAMs	URAMs	DSPs
Processor	670,720	537,162	1,080.5	673	2,527
⌊Platform	128,177	132,102	296.5	1	7
⌊Core	542,124	404,333	784	672	2,520
RPAU Unit	64,090	47,627	112	96	360
⌊Memory Core	13,754	999	96	96	–
⌊Dyadic Core	12,243	3,566	–	–	40
⌊Main Core	37,981	40,986	16	–	320
Butterfly Unit	1,545	1,592	–	–	10
⌊Modular Mult.	535	749	–	–	10
TF Gen. Unit	755	929	1	–	10

TABLE IV
RESOURCE UTILIZATION RESULTS ON ALVEO U250 CARD

$(N, \log_2 Q)$	LUTs	REGs	BRAMs	URAMs	DSPs
	(% utilization)				
$(2^{14}, 384)$	670,720 (39%)	537,162 (16%)	1,080.5 (40%)	673 (53%)	2,527 (20%)
$(2^{14}, 438)$	746,730 (43%)	581,731 (17%)	1,192.5 (44%)	769 (60%)	2,887 (23%)
$(2^{14}, 492)$	824,865 (48%)	637,381 (19%)	1,304.5 (48%)	865 (67%)	3,247 (26%)

we use only the half of available resources on the FPGA even for the design with the largest parameter set. It shows that we still have enough resources to instantiate extra arithmetic units for implementing more powerful architectures.

C. Comparison with related works

Notable works in the literature proposing efficient implementations for the high-level arithmetic operations of the HEAAN and other homomorphic encryption schemes in CPU [7], [16], [20], [27], GPU [3]–[6], [12], [18], FPGA [23], [33] and ASIC [14], [21], [22], [29] platforms.

Comparisons with SEAL: There are various highly-optimized software implementations of the HEAAN scheme based on homomorphic encryption libraries such as Microsoft SEAL [27] and Palisade [20]. We compare the performance of Medha with the single-threaded software implementation of the RNS-HEAAN on highly-optimized homomorphic encryption library Microsoft SEAL `v3.6` [27]. To present a fair comparison, we modified the SEAL accordingly to work with the parameter sets defined in Table I. The latency of high-level homomorphic operations in SEAL [27] and its comparison to Medha for Set-1, Set-2 and Set-3 are presented in Table V. The timing results of SEAL are obtained on an Intel i5-6200U CPU @ 2.30GHz × 4 with 16 GB RAM using `gcc` version 9.3 in Ubuntu 20.04.2 LTS. The proposed architectures with Set-1, Set-2, and Set-3 showed up to 79.1×, 73.7×, and 82.7× performance improvements, respectively, for high-level homomorphic operations compared to the SEAL-based implementation. The effectiveness of our approach increases with the larger parameter sets. In Table V, We also provide the performance results of SEAL running on a single-threaded Intel Xeon(R) Silver 4108 running at 1.80 GHz from the work [23] for the Set-2. Compared to this result, our architecture shows 137.8× speed-up for homomorphic multiplication and relinearization operations.

Comparisons with HEAX: The fairest comparison is with the HEAX processor [23]. It is the only prior art for the FPGA-based implementation of the RNS-HEAAN scheme. HEAX and Medha follow significantly different design methodologies. Unlike Medha, HEAX unrolls the key-switching of RNS-HEAAN into steps and then instantiates one dedicated block per step. These blocks are cascaded to realize a block-pipeline architecture. There are a total of six block-pipeline stages in the implementation of the key-switching operation. During a key switching, all the residue polynomials are processed one-by-one through the pipeline stages. Thanks to such unrolled and block-pipelined architecture, HEAX achieves a very high asymptotic throughput of 2,616 homomorphic multiplication including key-switching operations per second at 300MHz on a Stratix10 FPGA for the Set-2 parameter.

TABLE V
LATENCY COMPARISON WITH THE SEAL [27] AND HEAX [23]

	Work	Hom. Add	Hom. Mult. + Relin.
Set-1	Medha [1]	4.54 μs	353.64 μs
	SEAL [27]	359 μs (**79.1**×)	24,629 μs (**69.6**×)
Set-2	Medha	5.67 μs	483.70 μs
	SEAL [27]	418 μs (**73.7**×)	33,844 μs (**70.0**×)
	SEAL [23]	–	66,666 μs (**137.8**×)
	HEAX [23]	–	1,182.27 μs (**2.4** ×)
Set-3	Medha	5.67 μs	525.35 μs
	SEAL [27]	469 μs (**82.7**×)	39,143 μs (**74.5**×)

[1]: The implementation with 250MHz.

In comparison, Medha is an instruction-set architecture with programmability, and it reuses the building blocks again and again for computing different steps of various homomorphic routines. Naturally, Medha is a low latency-oriented architecture. It still achieves a competitive throughput (i.e., time/latency of one operation) of 2,067 homomorphic multiplications including key-switching operations per second while running at a lower clock frequency of 200MHz.

Latency-wise, Medha is more than 2× faster than HEAX as shown in Table V. As the latency figures of HEAX are not specified in [23], we estimate them based on the computation flow diagram from Table 5 and Figure 6 of [23] as follows. There are six stages of block-pipeline processing during a key-switching (the last row or Set-C of Table-5 in [23]) and the stages have been designed to have similar cycle counts. The first stage uses an 8-core inverse-NTT with at least 14,336 cycles latency. Thus, each stage has roughly 14,336 cycles of latency. As there are seven RNS-moduli and 18 pipeline stages including a one-core INTT stage with 114,688 cycles latency (see Figure 6 of [23]), computing a full key-switching will take at least 358,400 cycles. In comparison, our Medha has a latency of 96,740 cycles only for computing one homomorphic multiplication plus a key-switching.

We will discuss a bit more on the latency-vs-throughput figures. Thanks to the significantly lower latency, Medha would be advantageous for practical homomorphic applications compared to HEAX. The asymptotic throughput of HEAX is achievable only if we assume that in the application there are

plenty of data-independent homomorphic operations most of the time and that there is no overhead at the host side (e.g., a SW system) concerning managing the input-output ciphertexts. In real-life applications, there will be data dependencies and additionally, a SW host (which is running the application and using the HW as a service) will introduce some overhead in the processing of operand and result. Hence, the overall processing time of an application will greatly be determined by the latency instead of throughput.

There is one more advantage of using a low-latency system from a full-stack implementation point of view. Different homomorphic compilers have been designed to translate plaintext applications into homomorphic applications automatically. These compilers try to reduce execution time by reducing the number of homomorphic multiplications and depth of multiplication chains. If the latency is used as a 'cost'-metric, then the optimization task for a homomorphic compiler becomes simpler. On the other hand, if the accelerator is throughput-oriented, then the tasks for a homomorphic compiler become more challenging as it has to identify different ways of parallelization and also make necessary arrangements for handling the parallel ciphertexts (which are large in size).

We also note that Medha is somewhat smaller in area than HEAX. When the Set-2 parameter is considered, HEAX uses off-chip DRAM memory during the computations while Medha does not. Also, HEAX lacks flexibility and does not support programmability.

Comparisons with F1: A very recent concurrent work [14] has proposed a new instruction-based wide-vector processor F1. It supports high-level operations for BGV and HEAAN schemes in ASIC. It is primarily optimized for minimizing data movements during homomorphic computations. RTL synthesis of F1 in a commercial 14nm/12nm process reported an estimated chip area of 151 mm^2. Based on the cycle-accurate simulation, one homomorphic multiplication and relinarization operation of the HEAAN scheme requires only $2\mu s$ at a clock frequency of 1 GHz. A fair comparison between F1 and Medha is not possible as the two processors have been implemented on significantly different platforms. Medha has been designed considering the underlying FPGA-specific constraints. On the Xilinx U250 FPGA, the clock frequency of Medha is only 200 MHz which is almost $5\times$ slower than the frequency of F1 with one of the latest ASIC technologies. While we report accurate performance and resource consumption figures for Medha, the same for F1 will be available only after an F1 chip is made. Nevertheless, both works are instruction-set architectures and they show that such programmable and flexible architectures are potential candidates for HE architectures of the near future.

Comparisons with other HW implementations: The works in [25], [28], [31] present the FPGA implementations for the high-level operations of the BFV scheme. In [25], the authors proposed an implementation targeting vert large parameter set (namely $N = 2^{15}$ and $\log_2 Q = 1228$) and multiplicative depth. Their implementation suffers from heavy memory requirements as they need to continuously read and write DDR memory. In our architecture, we did not utilize any off-chip memory during the computations. In [28], Roy *et al.* uses smaller parameter set (namely $N = 2^{12}$ and $\log_2 Q = 180$) and shows $13\times$ performance improvement for homomorphic multiplication compared to the FV-NFLlib. Our design shows better performance and supports significantly larger parameter set. In [30], Turan *et al.* presents homomorphic encryption acceleration for the BFV scheme using the Amazon AWS FPGAs. It targets the same parameter set as [28] and can perform 613 homomorphic multiplications per second.

There are also works targeting acceleration of the BFV scheme using 'compute-in-memory' approach, where computations are performed using arithmetic units very close to the memory elements [22], [29]. This eliminates the data movements between memory and arithmetic units, and it reduces memory cost, which is a bottleneck for homomorphic encryption systems. However, near-memory arithmetic units can perform simple operations. As these works target accelerating the BFV scheme on a completely different platform, presenting a fair comparison between these works and Medha is not feasible.

Comparisons with GPU implementations: To the best of our knowledge, there are only two GPU implementations for the RNS-HEAAN scheme in the literature [4], [18]. Badawi *et al.* presents the first GPU-based implementation and they provide performance results for different parameters [4]. For the parameter set $N = 2^{14}$ and $\log_2 Q = 360$, they perform homomorphic multiplication and relinearization operations in 0.74 ms. For a similar parameter set (Set-1), our architecture shows $1.75\times$ better performance compared to their system running on an NVIDIA DGX-1 multi-GPU system with 8 V100 GPUs. The work in [18] uses a newer RNS variant of HEAAN scheme [17] and their implementation supports bootstrapping operation. They focused on memory-centric optimizations for the GPU platform, which is an NVIDIA Tesla V100, to implement operations of the HEAAN scheme efficiently. Their work targets very large parameter set, namely $N = \{2^{16}, 2^{17}\}$ and $\log_2 Q \approx 2300$. Therefore, it is not easy to perform a fair comparison between their work and our architecture. They claim to have $7.2\times$ performance improvement compared to the work of Badawi *et al.* [4].

VIII. Conclusions

Despite being theoretically sound, FHE suffers from performance issues due to its massive computational costs. In this paper, we proposed a programmable instruction-set architecture Medha for accelerating the cloud side operations of the RNS-HEAAN scheme. The accelerator gains its speed from parallel processing, efficient on-chip memory management, and other architectural design decisions. We experimentally tested the accelerator by running it on a Xilinx Alveo U250 card. Compared to the highly-optimized SEAL [27] library, our Medha achieves $137\times$ speedup on an Intel Xeon server. Furthermore, our results demonstrate almost $2.4\times$ performance improvement in the latency compared to the state-of-the-art hardware accelerator [23] for the same parameter. In this way, Medha pushes the limit of hardware acceleration for

cloud-side encrypted computations, and by doing so, it makes FHE practical for several privacy-preserving applications.

REFERENCES

[1] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," Cryptology ePrint Archive, Report 2015/046, 2015, https://ia.cr/2015/046.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "Towards the alexnet moment for homomorphic encryption: Hcnn, thefirst homomorphic cnn on encrypted data with gpus," 2020.

[4] A. A. Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "Privft: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, p. 226544–226556, 2020. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2020.3045465

[5] A. A. 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," Cryptology ePrint Archive, Report 2018/589, 2018, https://eprint.iacr.org/2018/589.

[6] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, pp. 70–95, May 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/875

[7] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52," 2021.

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," *Electron. Colloquium Comput. Complex.*, p. 111, 2011. [Online]. Available: https://eccc.weizmann.ac.il/report/2011/111

[9] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, ser. Lecture Notes in Computer Science, C. Cid and M. J. J. Jr., Eds., vol. 11349. Springer, 2018, pp. 347–368. [Online]. Available: https://doi.org/10.1007/978-3-030-10970-7_16

[10] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds., vol. 10624. Springer, 2017, pp. 409–437. [Online]. Available: https://doi.org/10.1007/978-3-319-70694-8_15

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[12] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

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

[14] A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, K. Eldefrawy, N. Genise, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption (extended version)," 2021.

[15] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford, CA, USA, 2009.

[16] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*. Springer, 2014, pp. 554–571.

[17] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," Cryptology ePrint Archive, Report 2019/688, 2019, https://ia.cr/2019/688.

[18] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 4, p. 114–148, Aug. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9062

[19] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, mar 2016. [Online]. Available: https://doi.org/10.1145/2893356

[20] Y. Polyakov, K. Rohloff, and G. W. Ryan, "Palisade lattice cryptography library user manual," *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep*, vol. 15, 2017.

[21] D. Reis, M. T. Niemier, and X. S. Hu, "A computing-in-memory engine for searching on homomorphically encrypted data," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, no. 2, pp. 123–131, 2019.

[22] D. Reis, J. Takeshita, T. Jung, M. Niemier, and X. S. Hu, "Computing-in-memory for performance and energy-efficient homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, p. 2300–2313, Nov 2020. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2020.3017595

[23] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: an architecture for computing on encrypted data," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

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

[25] 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 Transactions on Computers*, 2018.

[26] M. Scott, "A note on the implementation of the number theoretic transform," in *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*. Springer, 2017, pp. 247–258.

[27] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[28] 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 *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[29] J. Takeshita, D. Reis, T. Gong, M. Niemier, X. S. Hu, and T. Jung, "Algorithmic acceleration of b/fv-like somewhat homomorphic encryption for compute-enabled ram," Cryptology ePrint Archive, Report 2020/1223, 2020, https://ia.cr/2020/1223.

[30] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.

[31] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.

[32] Xilinx, "Xilinx DMA IP Reference Drivers," https://github.com/Xilinx/dma_ip_drivers.

[33] G. Xin, Y. Zhao, and J. Han, "A multi-layer parallel hardware architecture for homomorphic computation in machine learning," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[34] 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 Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 2, p. 49–72, Mar. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8544