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.

Fully Homomorphic Encryption (FHE) [27] 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. In 2009, Gentry constructed the first FHE scheme [17]. FHE quickly gained interest from both academia and industry. During the last 10 years better and better FHE schemes started appearing with orders of magnitude improvements in performances.

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'. In the remaining part of the paper we will use the terminology FHE to represent both classes. For evaluating arithmetic operations homomorphically, BFV [15] and BGV [9] are popular. THFE [12] is efficient for evaluating Boolean gates. For performing computations on encrypted *real numbers*, HEAAN [11] and its Residue Number System (RNS) variant RNS-HEAAN [10] are efficient. In fact, RNS-HEAAN is the fastest scheme for performing approximate computations on the encrypted *real* data, thus making it popular for privacy-preserving machine learning applications.

Although a decade of research in algorithmic and mathematical optimizations have made FHE schemes orders of magnitude faster than their first generation counterparts, homomorphic evaluations in software are five to six orders of magnitude slower than equivalent computations on plaintext. Therefore, hardware acceleration of FHE is crucial in reducing this performance gap. In the literature there are many works [16], [24], [25], [28], [33], [38] that implement selected building blocks of FHE or present simulation-based performance results. Often partial or simulation-based works produce overly optimistic performance results due to various assumptions. Therefore, real hardware accelerators are essential to accelerate FHE in practice and at the same time identify potential research directions for performance improvements. Surprisingly, there are only a few works that report real FHE accelerators [26], [29], [32], [34]. The accelerator [29] of 2018 could not make FHE faster than a typical software implementation. In 2019 the accelerator [32] achieved one order speed up with respect to the software. In the next year, 'HEAX' [26] obtained more than two order throughput with respect to the software. While the speedup is impressive, a limitation of HEAX is that it is not programmable and its block-pipelined architecture was designed specifically for the key-switching of RNS-HEAAN. In contrary to HEAX, the programmable accelerator [32] uses the same computational resources again and again as 'instructions' to execute a homomorphic encryption routine. While programmability is a desired feature in accelerators, the one order speedup of HEAX [26] over the programmable processor [32] may give us an impression that block-pipelined and specifically optimized accelerators are significantly superior to flexible accelerators for FHE. In this paper, we present a new programmable accelerator 'Medha' that reaches $2.4\times$ faster latency compared to HEAX and by doing so we bring

a new direction in the design space of accelerators for FHE.

Contributions: The main contributions of our paper reside at the high-levels of the implementation-hierarchy where different compute and memory elements are organized. To compute arithmetic of residue polynomials, we design a novel Residue Polynomial Arithmetic Unit (RPAU) pragmatically. Our RPAU contains a multi-core unified NTT unit for polynomial multiplication, two parallel sets of dyadic arithmetic units, and a customized on-chip memory for storing operand and resultant residue polynomials.

The designed RPAU is an instruction-set architecture. We can execute dyadic arithmetic and NTT instructions in parallel. This parallelism is very useful in minimizing the cycle count of key-switching operation, which is the costliest subroutine in FHE. We observe around 40% reduction in the latency at the cost of around 20% increase in the area.

A memory conservative design approach is followed to save on-chip memory elements for useful computations. A customized on-chip memory is designed to store residue polynomials inside the RPAU. Even for a large polynomial size with 2^{14} coefficients, the on-chip memory is able to store all the residue polynomials during a homomorphic multiplication and key-switching, therefore eliminating the need for any off-chip data exchange during a computation (which is very slow). We are the first to report fully on-chip computation of the two FHE subroutines for such large-degree polynomials.

At the next level of the implementation hierarchy, multiple RPAUs are instantiated and interconnected. RPAU-to-RPAU data exchanges happen during key-switching and rescaling subroutines of FHE. Finding an optimal way of interconnecting the RPAUs is critical due to two reasons. Firstly, because each RPAU consumes a large area and has thousands of bits of input/output ports, their placement on the design-platform becomes a challenging engineering problem. Several works, e.g., [28], [29] bypassed that engineering problem by assuming data exchanges happen via a shared off-chip memory. Secondly, a slow data exchange will have a drastic impact on the performances of the key-switching and rescaling subroutines. We studied different ways of interconnecting the RPAUs and found that a ‘Ring’ interconnection is the most optimal in reducing the number of RPAU-to-RPAU wires. During the key-switching and rescaling, data exchanges between RPAUs happen through the ring without any hazard. The proposed ring is crucial in overcoming SLR-to-SLR wire limitations in large Xilinx FPGAs and making placement feasible.

Besides the above-mentioned main contributions, we make optimizations at the lower levels of the implementation hierarchy where polynomial and coefficient operations are performed. We implement a unified and multicore NTT-multiplier with optimal scheduling for memory reads and writes. We use hardware-friendly parameters (e.g., word-size, primes, etc.) and parallel algorithms to perform fast modular arithmetic.

The paper is organized as follows. Sec. II presents a brief mathematical background that will be useful to understand the paper. Next, the accelerator architecture is realized hierarchically starting from low-level polynomial arithmetic units in Sec. III and then organizing different compute and memory elements in Sec. IV. Our main contributions are in Sec. IV. Detailed experimental results and comparisons are provided in Sec. V. The final section draws the conclusions.

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 [9], BFV [15], and HEAAN [10]. 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 [10]. 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 [11].

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 [10]. To use RNS-HEAAN in an application, the first step will be to set up

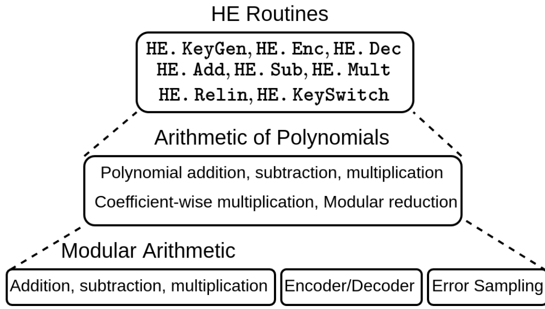

Fig. 1. Implementation hierarchy of homomorphic encryption. Key generation, encryption, and decryption are performed on the user side. These user-side operations also include encoding, decoding, and Gaussian sampling at the lowest level. Homomorphic evaluations on ciphertexts using HE. Add, subtract, 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.

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 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 modular arithmetic.

TABLE I
PARAMETER SETS

Param. Set	$(N, \log_2 Q)$	$L+1$	Mul. Depth	Sec. 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].

Parameter set for our implementation: To implement residue number system (RNS)-based FHE, we use 60 and 54-bit prime moduli similar to SEAL [31] and HEAX [26]. By increasing or decreasing the number of moduli in the RNS, the multiplicative depth can be adjusted. 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.

With these parameter sets, 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 [21], etc.

In the next several sections, we describe how to implement the levels of the pyramid of Fig. 1 starting from the lowest level and then moving up gradually.

III. Implementation of low-level arithmetic

The lowest level of the implementation hierarchy (Fig. 1) contains modular arithmetic, namely addition, subtraction and multiplication. They are the most frequently used modules. We use bit-parallel multipliers made of DSPs available in FPGAs. To bypass the expensive modular reduction circuits from [26], [32], we use pseudo-Mersenne primes in the RNS base and perform very cheap modular reductions. Similar reduction circuits are popular in lattice-based post-quantum cryptography.

The next level of Fig. 1, performs the arithmetic of large-degree polynomials. Polynomial multiplication is the most expensive operation and they are computed in $O(n \log n)$ time complexity using the Number Theoretic Transform (NTT) method. Our NTT-based multiplier uses the decimation-in-time (DiT) approach for the forward NTT and decimation-in-frequency (DiF) approach for the inverse NTT (INTT) [30].

A. Parallel NTT architecture

For fast polynomial multiplications, we implement a multi-core parallel NTT unit borrowing the best practices from [26], [29], [32] including the routing and BRAM access optimizations from [29]. In addition, we apply an 'address delaying'

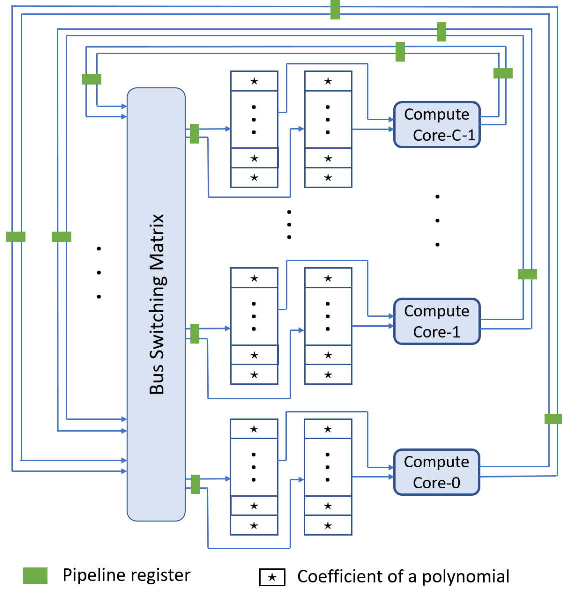

Fig. 2. Organization of memory and compute cores for efficient implementation of parallel NTT

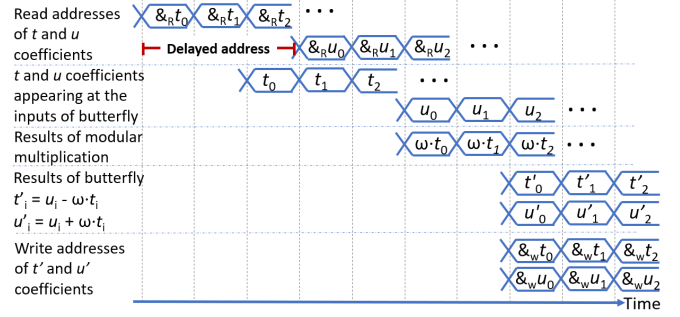

Fig. 3. The timing diagram for our DiT method of NTT. Due to pipelined datapath, the reading of the u coefficients is delayed so that we can add or subtract them when the corresponding modular multiplication results $t \cdot \omega$ are ready. The results of butterfly operations are written synchronously. 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.

technique that results in a significant reduction in the register consumption without causing any performance overhead. Fig. 2 shows a high-level organization of the memory elements (for storing the polynomial-parts) and the butterfly cores (for processing the coefficients) inside the NTT unit. The bus matrix rearranges the processed coefficients before writing them to the memory elements. We put multiple layers of pipeline registers (shown in green in Fig. 2) to increase the clock frequency. 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.

B. Unified butterfly core for DiT NTT and DiF INTT

We design a single NTT unit for the DiT NTT and DiF INTT. We propose an 'address delaying' optimization that causes significant reduction in the register consumption of the pipelined butterfly cores. 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 $(u'', t'') \leftarrow (u + t, (u - t) \cdot \omega)$. Our unified butterfly core uses one modular multiplier, one modular adder, two modular subtractors and a few two-to-one multiplexers. The modular multiplier is heavily pipelined (around 20 stages) to keep the clock frequency over 300 MHz.

Now we describe the address delaying optimization. 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 [29] both u and t are read together from the memory and then u is passed through a long chain of redundant registers just to make sure that both u and $\omega \cdot t$ arrive together at the adder and subtracter.

Our pipeline strategy avoids the above-mentioned bloated register consumption and saves 153K registers 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. 3 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 (equal to the number of pipeline stages in the modular multiplier) ahead of reading the u-coefficients. As a consequence, each modular multiplication result and the respective u appear synchronously at the inputs of the adder and subtractor circuits for correct computation.

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

Optimized software implementations (e.g., SEAL [31]) and also the hardware implementations [26], [32] save cycle count of NTT by keeping all twiddle constants in large tables. As previous hardware implementations [26], [32] indicated that FHE is memory-bound, we follow a memory-conservative design approach and compute the twiddle constants on the fly and in parallel to the butterfly operations, therefore avoiding large BRAM consumption.

A twiddle factor generation unit mainly consists of a modular multiplier and one memory for keeping a few initial constants. The multiplier is not an additional cost as it is reused to parallelize coefficient-wise multiplication and modular reduction during dyadic operations.

IV. Architecture of the Homomorphic Processor

We take the optimized polynomial arithmetic units from the previous section and organize them to compute homomorphic subroutines, namely homomorphic addition/subtraction, multiplication, key-switching, and relinearization. Medha is an instruction-set architecture (ISA) and therefore programmable

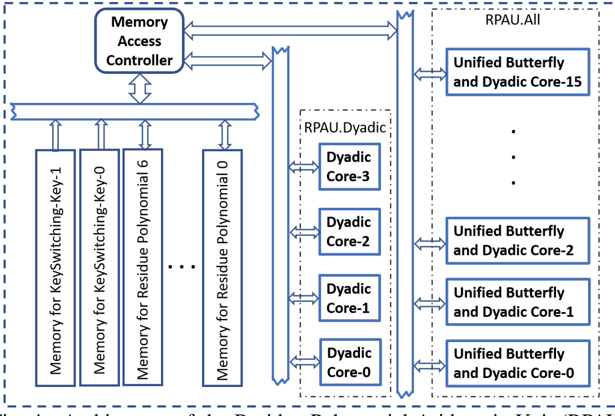

Fig. 4. Architecture of the Residue Polynomial Arithmetic Unit (RPAU).

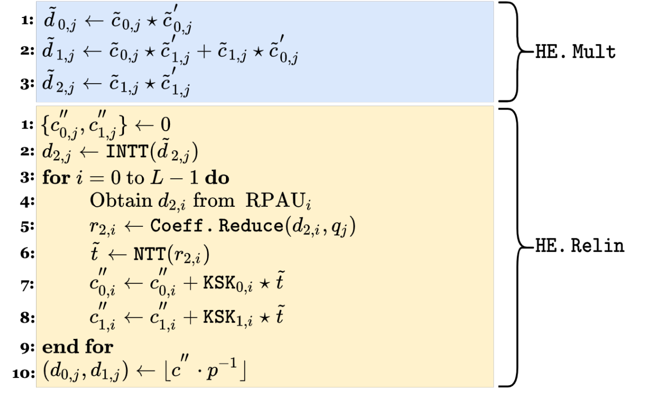

Fig. 5. Computation steps that are performed in the j-th RPAU during a homomorphic multiplication followed by a key-switching operation. The tilde is used to indicate that a data variable is in the NTT domain. Coefficient-wise multiplication of two polynomials is denoted using $\star$.

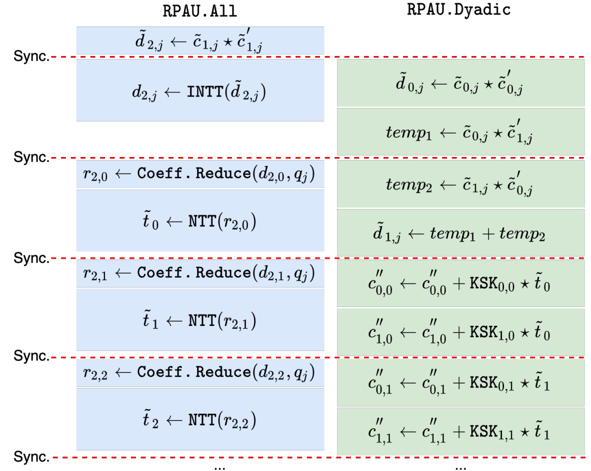

Fig. 6. Parallel processing of Fig. 5 using two threads inside an RPAU.

to run the homomorphic subroutines flexibly by reusing the same polynomial arithmetic units many times. Previous works [29], [32] presented ISAs for accelerating homomorphic encryption, but their performance advantages remained limited, e.g., only one order speedup [32] compared to SW implementations. On the contrary, HEAX [26] organized its polynomial arithmetic elements in a specific block-pipelined manner to realize a key-switching unit that gives over two order higher throughput compared to SW. The one order speedup of HEAX over [32] demonstrated that block-pipelined and specific FHE accelerators are much superior to programmable accelerators that reuse compute elements. We organize compute and memory elements pragmatically and realize a *programmable accelerator* that achieves more than 2× faster latency compared to HEAX. Our work shows that FHE accelerators do not have to sacrifice programmability to achieve high speed. The following subsections describe how we organize compute and memory elements.

A. Design of Residue Polynomial Arithmetic Unit (RPAU)

In any RNS-based FHE, an arithmetic operator is applied to a 'vector' of residue polynomials. The idea has some similarities with the Single Instruction Multiple Data (SIMD) processors. To benefit from such arithmetic parallelism [28], Medha instantiates multiple high-level units for processing the residue polynomials in parallel. These units are called the Residue Polynomial Arithmetic Unit (RPAU) and they are ISA. Any high-level instruction for Medha essentially gets translated into instructions for the RPAUs.

Fig. 4 shows the organization of polynomial arithmetic cores and memory elements inside our proposed RPAU. We observe that the inner loop of key-switching or re-linearization (see Sec. II) executes one NTT and several coefficient-wise polynomial operations. Therefore, we keep two groups of compute cores, namely `RPAU.All` and `RPAU.Dyadic` in the RPAU. The `RPAU.All` group is capable of performing all kinds of polynomial arithmetic operations fast using 16 cores. The `RPAU.Dyadic` group can only perform coefficient-wise (i.e., dyadic) operations using only 4 cores. The two compute groups are executed in parallel during key-switching (or re-linearization) and re-scaling (or mod-down) operations.

Parallel execution of `RPAU.All` and `RPAU.Dyadic`: Our RPAU can execute two instructions, one using `RPAU.All` and the other using `RPAU.dyadic`, concurrently in parallel and save around 40% cycles. The rationale behind this novel design decision is explained as follows.

Fig. 5 shows data dependencies between the steps during homomorphic multiplication and key-switching. We see that some of the steps, e.g., $d_{0,j}$, $d_{1,j}$, and $d_{2,j}$ in the first block can be performed in parallel. Inside the loop of the key-switching, the steps are sequential due to data dependencies. As the loop iterates several times, we can unrolling it and then 'block-pipeline' the loop-internals. We have different options for applying parallel processing.

- Option 1: Running more than one NTTs in parallel inside the RPAU will be useful if we unroll the key-switching loop in Fig. 5. E.g., unrolling by a factor 4 will require 4 NTT units, therefore almost increasing the logic count of RPAU by 4 times. Hence, this approach is not attractive.
- 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: Use only one NTT unit in the RPAU and reduce or completely hide the latency of the coefficient-wise operations by executing NTT and coefficient-wise polynomial operations in parallel. Using a few extra modular adder and multipliers, we can compute these cheap coefficient-wise operations in parallel to NTTs. For example, using only four extra modular arithmetic cores, we can compute two dyadic polynomial arithmetic instructions (taking $2 \times 4,096$ cycles) concurrently to an NTT (taking around 7,200 cycles) and effectively reduce the latency of steps 6-8 in Fig. 5 to the latency of one NTT only.

We apply Option 3 as it is computationally fast and at the same time requires a minor increase in the logic area. The `RPAU.Dyadic` group in Fig. 4 executes coefficient-wise instructions in parallel to NTT instructions in the `RPAU.All` group. We observe 40% reduction in the latency at the price of only 20% increase in the logic resources. Fig. 6 gives a timing diagram and shows how we can speedup the computation of Fig. 5 using the two parallel compute groups.

B. Organization of on-chip memory inside RPAU

The polynomials operands in FHE are large (one ciphertext is 1.7 MB for Set-2). All previous FPGA implementations required expensive off-chip data exchanges. We are the first to report fully on-chip memory based execution of homomorphic multiplication, key-switching and re-scaling. The amount of accessible on-chip memory in our FPGA is limited. Therefore, we followed memory-conservative design approach for the low-level polynomial operations, e.g., on-the fly generation of twiddle constants (Sec. III-C) instead of large tables and in-place NTT. In the end, we found that we have sufficient BRAMs and URAMs in U250 FPGA to keep all the operand ciphertexts and large key-switching keys on-chip. On the contrary, HEAX [26] used large tables and split-BRAMs for its NTTs, and dedicated BRAMs for each stage of its block-pipelined architecture. As a consequence, for Set-2 with $N = 2^{14}$, HEAX needs to perform off-chip data transfer during key-switching. In the following part, we describe how we combine available BRAMs and URAMs to realize RPAU's on-chip memory unit.

Peak memory requirement: Optimizing the steps of homomorphic multiplication + key-switching, we observe that the peak memory per RPAU is equal to storing seven residue polynomials (ciphertext-dependent data), and additional $2L$ residue polynomials for the key-switching-key. E.g., for Set-2 we have $L = 7$ and therefore we need to store 21 residue polynomials on-chip to eliminate off-chip memory access during key-switching. Neither BRAMs nor URAMs alone in U250 could such a big data, but their combination can.

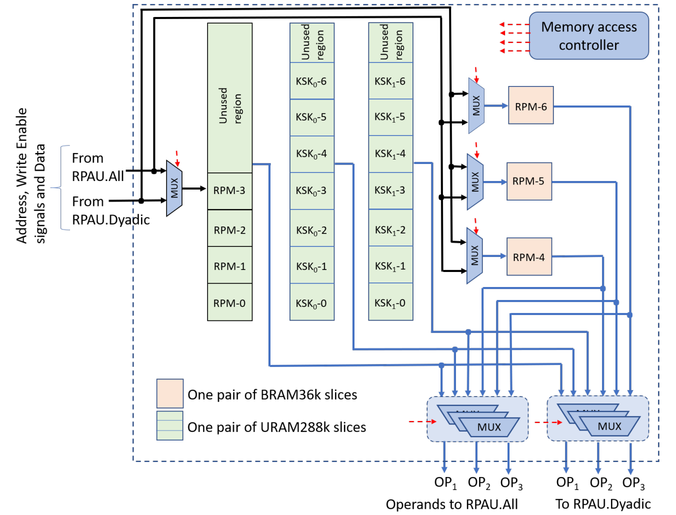

Fig. 7. Organization of memory elements inside the memory bank. One memory bank is connected to a one core of the NTT unit. There are 16 such memory banks inside each RPAU. Any residue polynomial is split into 16 fragments, and one fragment is stored in one RPM-i of a memory module.

Memory unit: Quantitatively, one residue polynomial needs 32 BRAM36k or 4 URAM slices. If a polynomial is stored using 4 URAMs, then due to URAM's i/o port limitation we cannot use all 16 cores of the NTT unit. Hence, designing the memory unit of the RPAU requires careful considerations of the computation and architectural constraints.

We make the memory organization modular inside the RPAU. To provide fast data access during NTT, we assign one 'memory-bank' exclusively to each butterfly core. One 'memory bank' is a heterogeneous collection of BRAMs and URAMs as shown in Fig. 7. The memory-bank of the i-th core of NTT keeps only the i-th fragments of all the 21 residue polynomials. In Fig. 7 the abbreviation 'RPM' stands for the residue polynomial memory. There are seven RPMs as there are seven ciphertext-dependent residue polynomials. 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. Programmer decides which polynomial goes to which RPM taking care of data dependencies and access patterns of the FHE subroutine. The 'Memory access controller' block is responsible for handling memory accesses of the two parallel computing groups `RPAU.All` and `RPAU.Dyadic`.

C. Interconnecting multiple RPAUs

The designed RPAU from the previous section is a fundamental compute element in our programmable architecture.

Just using one RPAU in a time-shared manner we can process an FHE subroutine. For fast computation time, we instantiate several RPAUs in parallel. As key-switching and re-scaling operations require exchanging the residue polynomials for different moduli, the RPAUs must perform data exchanges between them. If performed via a shared memory, such data exchanges will slow down the FHE operations. On the other hand, connecting the RPAUs in the form of a star network will bring placement hurdles or may even make an actual HW implementation impossible on FPGAs. We present a novel way of connecting the RPAUs and optimally solve the problem.

First, we introduce the challenges in interconnecting the RPAUs and then describe our solution. Like other large-scale Xilinx FPGAs [36], our Alveo U250 platform uses SSI technology and consists of four 'semi-separated' SLR regions. Two neighbouring SLRs are connected using a limited number of wires. We observe that one SLR could fit up to three RPAUs and hence at least three SLRs are needed to implement Medha. Interconnecting the RPAUS must take SLR-SLR connection constraint into account to make implementation feasible.

Additionally, the communication unit between of the FPGA resides in SLR1. An input ciphertext should be sent efficiently and stored in the memory blocks of the RPAUs that reside in the other SLRs. Similarly, output polynomials should be read from different SLRs to SLR1.

The naive solution would be connect the RPAUs in a 'Star' network keeping separate paths for each connection. We found that such an interconnection complicates the routing, bolts the number of nets crossing the SLRs, and ultimately reduces the clock frequency to around 50 MHz or less.

We propose a 'Ring' interconnection of the RPAUs to reduce the routing: only two neighbouring RPAUs are connected. In many general-purpose applications, a ring network is considered slow due to its serial communication. Interestingly, after analyzing the computation steps of key-switching and re-scaling, we find that the exchange of residue polynomials between the RPAUs could be transformed into a *broadcasting* protocol where only one RPAU broadcasts a polynomial at a time and all the remaining RPAUs receive. Such a transformation does not add any computation overhead.

Therefore, in Medha we connect neighbouring RPAUs only and the arrangement of all the RPAUs looks like a ring. Each RPAU sends its data to any other RPAU through a chain of RPAUs. In Fig. 8 we show placement of RPAUs on the floor of the FPGA. The ring is marked with a red line.

For external communication signals, we followed the 'ring' connection for sending data signals from SLR1 to other SLRs as shown in Fig. 8.

D. 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

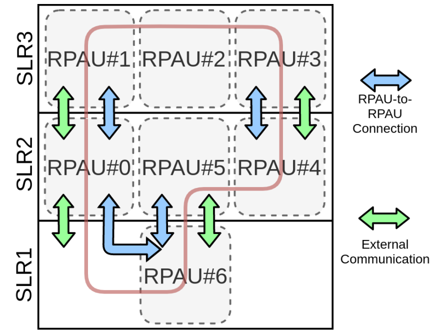

Fig. 8. The proposed 'ring' structured floorplan to minimize routing cost for the implementation with 7 RPAU units.

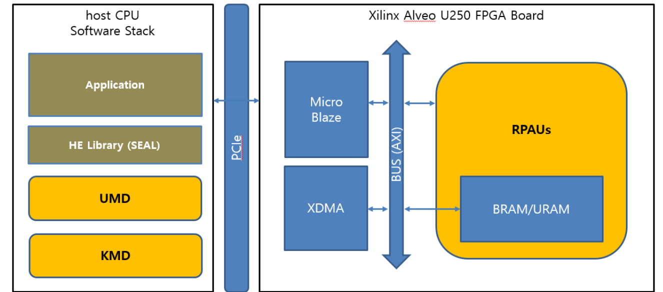

Fig. 9. CPU-FPGA interface and software stack

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.

E. Hardware-Software Interfacing of the Overall System

We implemented a proof-of-concept software stack (Fig. 9) 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 [37] for fast transfers. We use the MicroBlaze (Xilinx's microprocessor) unit for controlling the communication between the host CPU and the RPAUs, and also for monitoring the entire FPGA system.

F. Implementations on other platforms

Although we implemented Medha in Alveo U250, the architecture is not tied to the specific FPGA. Large Xilinx FPGAs use the SSI technology [36] to combine multiple Super

Logic Regions (SLR). Therefore, the proposed ring of RPAUs (Sec. IV-C) will be essential to implement multi-RPAU FHE architecture on large Xilinx FPGAs.

Multi-FPGA implementation of Medha will enhance its performance further and enable implementations for larger parameter sets. Each FPGA will host a set of RPAUs. Bootstrapping will definitely need a stack of large FPGAs due to its sheer complexity. As FPGA-FPGA communication faces wiring constraints similar to SLR-SLR communication, the proposed ring interconnection of RPAUs can be extended to the multi-FPGA setting by connecting the FPGAs in a ring. We consider developing a multi-FPGA implementation of Medha as a future work as we anticipate such work will also take a long design time, a year or more.

Medha can be ported to ASIC technologies. Medha has been described using Verilog RTL. Only the BRAMs/URAMs and DSP multipliers are Xilinx IPs. In ASIC, BRAMs and URAMs will be replaced by dual-port SRAMs with the same functional and timing behavior. Similarly, the DSP multipliers will be replaced by normal multipliers. Interesting, the proposed ring interconnection of RPAUs will be ideal for an ASIC implementation. In the ring, neighboring RPAUs will be placed side by side on the layout of the chip. We anticipate that Medha's clock frequency will improve by three to five times depending on the ASIC technology.

V. Results

We described Medha in Verilog HDL and implemented in Xilinx Alveo U250 card. We used 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. The cycle counts were collected using a hardware-based counter. 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.

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.
All cycle counts are reported using a hardware-based counter.

Speed of homomorphic multiplication and relinearization: As Table II reports only the total cycle counts obtained from the HW counter, we give an explanation of the count values. The `HE.Mult` operation is performed using coefficient-wise multiplication and addition instructions. `HE.Relin` which is a key-switching operation, is used to linearize the result of `HE.Mult`. As shown in Fig. 5, `HE.Relin` requires each $\tilde{d}_{2,j}$ polynomial in modulo q_j to be reduced by the other moduli. Hence, all $\tilde{d}_{2,j}$ polynomials are transformed to the time-domain first using one INTT instruction in the parallel RPAUs. Then, the j-th RPAU boradcasts the polynomial $d_{2,j}$ to the other RPAUs through the ring. Received $d_{2,j}$ are modulo-reduced by the other moduli and then converted back to the NTT domain. This procedure is repeated for $j = 0, 1, \ldots, L-1$ as shown in Fig. 5. Note that, the coefficient-wise multiplications and accumulations in the key-switching loop are performed in parallel to the NTTs using `RPAU.Dyadic`. Due to such parallel processing, the overall cycle count of `HE.Relin` is primarily determined by the latency of computing 1 INTT followed by L NTTs. Therefore, using the cycle counts of NTT and INTT from Table II, we can estimate that `HE.Relin` will take a couple of thousands more than 51K cycles for Set-1 ($L = 6$), 58K cycles for Set-2 ($L = 7$), and 65K cycles for Set-3 ($L = 8$). The accurate cycle counts in Table II are larger as they include ciphertext multiplication, `HE.Rescale`, and some additional dyadic operations that do not occur in parallel to the NTTs. The cost of `HE.Rescale` is explained next.

Speed of homomorphic rescaling: `HE.Rescale` is used to rescale a ciphertext. It takes the two residue polynomials of the extra RNS modulus, e.g., the special modulus p, and then converts them to the time domain by applying two INTTs. Then, the two polynomials are broadcast by its RPAU through the ring. The remaining RPAUs reduce the two polynomials and then convert them back to the NTT domain executing two NTT instructions. Then coefficient-wise scaling takes place, and finally the resultant polynomials are added to the ciphertext polynomials of the remaining moduli. Therefore, we see that the latency of `HE.Rescale` is primarily the cost of executing 2 INTTs and 2 NTTs.

But we do better than that by overlapping the second INTT of the extra residue polynomial with the first NTT in the remaining RPAUs. Due to that optimization, a total of $\approx$26,208 clock cycles are required for the `HE.Rescale` operation.

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%)

B. Resource Utilization results

Table III shows the resource requirements for the complete processor architecture (with Set-1), one RPAU, one butterfly core, and one twiddle factor generation core. The area figures for the complete processor include the 'Platform' unit that is responsible for the communication between the FPGA and the host CPU.

Table IV shows the resource requirements for all three designs (Set-1, 2, and 3). Excluding the URAM utilization, we use approximately half of the available resources in Alveo U250. We followed a conservative design approach to ensure that at the completion of the project the entire processor fits in the FPGA with a high probability. If more design effort is added, it might be possible to reduce the cycle count further by instantiating more compute cores, e.g., 32 butterfly cores. However, there can be a reduction in the clock frequency due to the increased complexity.

Power consumption: Alveo U250 FPGA card can have a 215W maximum power consumption level [35] with a typical consumption level of around 100W. For Medha, Vivado reported on-chip power consumption of 55.3W, 50.6W and 55.2W for Set-1, 2 and 3 respectively. Although the implementation with Set-1 has fewer RPAUs, it runs at a higher clock frequency (250 MHz unlike 200 MHz) and therefore reports a higher power consumption compared to Set-2 and 3.

C. Benchmarking of homomorphic applications

Transforming plaintext applications into *optimized* homomorphic applications is generally not straightforward [14]. In this work, we focused on the processor architecture design and considered FHE application development and benchmarking as future works. Similar to HEAX [26] and F1 [16], we provide estimates of application benchmarking.

Our parameter sets are sufficient for evaluating logarithmic and exponential functions which are used for approximate computations [20]. PrivFT implementation [4] proposed for text classification using homomorphic encryption runs a shallow network consisting of an embedding (hidden) layer and an output fully connected layer. For the parameter $N = 2^{14}$ and $\log_2 Q = 360$ (i.e., $L = 5$), PrivFT inference takes a ciphertext and performs three multiplications with scalars, 61+14=75 additions, and 14 rotations. Medha with Set-1 can perform the same operation in 0.36 seconds and achieve $1.8\times$ speedup over the GPU implementation of PrivFT [4].

In [8], the privacy-preserving forecasting application in the smart grid scenario uses a GMDH network having three hidden layers. It is evaluated for a smaller parameter (namely $N = 2^{12}$ and $\log_2 Q = 186$) with $L = 3$). Medha's Set-1 parameter is overly larger than theirs, yet Medha performs this evaluation $150\times$ faster. Every half an hour, Medha can evaluate the energy price forecast for $60,000$ apartments, which is sufficient for a small city, thus making Medha useful in the real-world scenario.

D. Comparison with related works

Comparisons with SEAL: There are various highly-optimized software implementations of the HEAAN scheme based on homomorphic encryption libraries such as Microsoft SEAL [31] and Palisade [23]. 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` [31]. 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 [31] 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 from a single-threaded Intel Xeon(R) Silver 4108 running at 1.80 GHz, which was used in the HEAX paper [26]. Compared to this result, Medha shows 137.8× speed-up for homomorphic multiplication and relinearization operations.

Intel HEXL [7] uses AVX-512 intrinsics and gains around 1.8× speedup over SEAL. For Set-2, on an Intel i9-7900X CPU running at 3.3 GHz, results show that Medha provides 28.5× speed-up for homomorphic multiplication and relinearization. While FPGA prototyping is the first step, an ASIC implementation of Medha will increase (anticipated 3 to 5× or more depending on technology) the operating frequency of Medha and thereby increase the ratio of hardware acceleration.

Comparisons with HEAX: The fairest comparison is with the HEAX processor [26]. It is the only prior art for actual FPGA-based implementation of RNS-HEAAN and also with

TABLE V
LATENCY COMPARISON WITH THE SEAL [31] AND HEAX [26]

	Work	Hom. Add	Hom. Mult. + Relin.
Set-1	Medha [1]	4.54 μs	353.64 μs
	SEAL [31]	359 μs (**79.1**×)	24,629 μs (**69.6**×)
Set-2	Medha	5.67 μs	483.70 μs
	SEAL [31]	418 μs (**73.7**×)	33,844 μs (**70.0**×)
	SEAL [26]	– –	66,666 μs (**137.8**×)
	HEAX [26]	– –	1,182.27 μs (**2.4** ×)
Set-3	Medha	5.67 μs	525.35 μs
	SEAL [31]	469 μs (**82.7**×)	39,143 μs (**74.5**×)

[1]: The implementation with 250MHz.

the same parameter set (Set-2). 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 to attain high throughput. 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.

In comparison, Medha is an instruction-set architecture with programmability, and it reuses the RPAUs 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 [26], we estimate them based on the computation flow diagram from Table 5 and Figure 6 of [26] as follows. There are six stages of block-pipeline processing during a key-switching (the last row or Set-C of Table-5 in [26]) 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 [26]), 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.

From an architect's perspective, Medha has four main advantages over HEAX: (i) Medha shows better latency performance with a competitive throughput, (ii) Medha uses only on-chip memory during the computations while HEAX needs off-chip memory communication during the key switching operation for Set-2, (iii) HEAX uses a fixed-pipelined architecture tailored for HEAAN scheme while Medha's instruction-based architecture allows flexibility, and (iv) HEAX uses separate arithmetic units to perform homomorphic multiplication and relinearization, and it does not perform rescaling, homomorphic addition and subtraction operations in hardware (see Fig. 7 in [26]). We should also note that the design principles of Medha were not selected to outperform HEAX. Our main goal was to gain fast computation time while keeping programmability. Our results show that our instruction-set architecture can outperform a block-pipelined and specific architecture in terms of latency. Our results bring a new direction to the design space, otherwise HEAX's one order superiority over the previous programmable processors [29], [32] would have indicated that specific and block-pipelined processors are the way to accelerate FHE in HW.

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. Note that, already due to the batching of messages, a homomorphic operation implicitly performs the operation on $N/2 = 8,192$ slots in parallel. 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).

Comparisons with F1: Concurrent to our work, [16] proposed an instruction-based wide-vector processor architecture 'F1' and presented RTL simulation-based performance estimates. In 14nm/12nm process, RTL synthesis of F1 reported an estimated chip area of 151 mm^2 and 1 GHz operating frequency. F1 achieves a very high throughput thanks to its fully unrolled and pipelined arithmetic units. They report a throughput of 500,000 homomorphic multiplication and relinearization per second at 1 GHz for Set-2 parameter. However, latency figures

for homomorphic operations are not presented in F1.

In this paragraph, we present a normalized performance comparison between F1 and Medha. F1 has 16 high-level clusters, each having one 'unrolled' NTT and many dyadic units. The unrolled NTT uses a condensed 2D array of butterfly circuits. The number of multipliers in one cluster of F1 is roughly 50% of the total multipliers available in one Alveo U250 board. If we assume that the FPGA keeps one cluster of F1 and runs at 200 MHz, then the performance of F1 will drop by 80 times (16 times due to the use of only one cluster and another 5 times due to slower frequency), excluding the overhead of off-chip access. In this scenario, F1 and Medha will have almost the same speed. If the overhead of off-chip access is considered, then F1 will be slower than Medha. In reality, F1 may not be implementable in present-day FPGAs. Unrolled NTT was first proposed in [18] and was found to be impractical in FPGAs. As the multipliers are distributed homogeneously on the FPGA floor, and SLR-to-SLR connections are limited in number, it is likely that F1's unrolled NTT may not fit in FPGAs. The authors of F1 did not report any FPGA-based results.

There are several additional limitations of the F1 architecture. Firstly, F1 uses 32-bit moduli whereas Medha uses 60 and 54-bit moduli similar to SEAL. Thus F1 supports lower-precision arithmetic of encrypted real numbers than Medha and SEAL. With 32-bit moduli, Medha could accommodate around 16 RPAUs and therefore support around 15 multiplicative levels. Secondly, unlike Medha's efficient ring structure, F1 uses a crossbar network for realizing cluster-to-cluster communication. The proposed crossbar network uses a 10 mm^2 chip area with 19.6W design power. The Medha's ring structure limits the direct communication of an RPAU to only its two neighboring RPAUs, hence it reduces the number of interconnects significantly.

The biggest limitation of F1 is that there is no real ASIC implementation of it. Its simulation-based performance estimates will be impressive only if a real F1 chip is built. Authors of F1 consider chip-simulation as future work. Bypassing FPGA-based prototyping, fabricating a silicon chip from the RTL of F1 will involve major engineering.

Comparisons with other HW implementations: The works in [29], [32], [34] present the FPGA implementations for the high-level operations of the BFV scheme. In [29], the authors proposed an implementation targeting vert large parameter set (namely $N = 2^{15}$ and $\log_2 Q = 1228$) and their implementation suffers from off-chip memory communication requirements. The works in [32] and [34] use smaller parameters (namely $N = 2^{12}$ and $\log_2 Q = 180$) and shows performance improvements for homomorphic multiplication operation compared to the FV-NFLlib. Our design shows better performance and supports significantly larger parameter set.

There are simulation 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 [25], [33]. As these works use a significantly 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], [19]. For the parameter set $N = 2^{14}$ and $\log_2 Q = 360$, [4] performs homomorphic addition and homomorphic multiplication plus relinearization in 0.04 ms and 0.74 ms, respectively. For a similar parameter set (Set-1), our architecture shows $8.8\times$ and $2.1\times$ better performance compared to their system running on an NVIDIA DGX-1 multi-GPU system with 8 V100 GPUs for homomorphic addition and homomorphic multiplication plus relinearization, respectively. Also, their multi-GPU platform NVIDIA DGX-1 has a reported maximum power consumption level of 3,500W which is $16\times$ higher compared to the Alveo U250 board. The NVIDIA DGX-1 platform costs $49,000 which is $6.3\times$ higher than the Alveo U250 board (at peak power). Therefore, Medha running on an Alveo U250 board would be a more power-efficient and cheaper accelerator solution for homomorphic applications.

The work in [19] supports bootstrapping and focuses on memory-centric optimizations for an NVIDIA Tesla V100 GPU. 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. There are also other GPU-based accelerator implementations in the literature targeting other HE schemes (i.e., BFV) [3], [5], [6], [13] or partial operations such as NTT [22], [39].

VI. Conclusions

Despite being theoretically sound, FHE suffers from performance issues due to its massive computational costs. Medha is a programmable accelerator architecture that has been designed pragmatically to overcome the speed limitations of previous FHE accelerators. Medha gains efficiency from its highly optimized polynomial arithmetic blocks, parallel processing of instructions inside RPAU, parallel processing of residue polynomials using many RPAUs, highly efficient ring-based interconnection of RPAUs for data exchange, and customized on-chip memory unit. The memory-conservative design approach used to build Medha made it possible for the first time to compute key-switching without requiring any off-chip communication even for a large FHE parameter.

Medha was implemented in the Xilinx Alveo U250 card and accurate performance results were obtained. Compared to the highly-optimized SEAL [31] library, our Medha achieved $137\times$ speedup on an Intel Xeon server. Medha achieved almost $2.4\times$ latency improvement compared to the block-pipelined and specific hardware accelerator HEAX [26]. The speed improvement shows that hardware accelerators for FHE can attain high performance without losing programmability or flexibility. Performance benchmark results for two inference applications on encrypted data showed that Medha makes FHE practical for several privacy-preserving applications, at least for inference calculations.

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] J. W. Bos, W. Castryck, I. Iliashenko, and F. Vercauteren, "Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling," in *Progress in Cryptology - AFRICACRYPT 2017 - 9th International Conference on Cryptology in Africa, Dakar, Senegal, May 24-26, 2017, Proceedings*, ser. Lecture Notes in Computer Science, M. Joye and A. Nitaj, Eds., vol. 10239, 2017, pp. 184–201. [Online]. Available: https://doi.org/10.1007/978-3-319-57339-7_11

[9] 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

[10] 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

[11] 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

[12] 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.

[13] 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.

[14] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 142–156. [Online]. Available: https://doi.org/10.1145/3314221.3314628

[15] 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

[16] 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.

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

[18] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss, "On the design of hardware building blocks for modern lattice-based encryption schemes," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 512–529. [Online]. Available: https://doi.org/10.1007/978-3-642-33027-8_30

[19] 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

[20] S. Khanna and C. Rafferty, "Accelerating homomorphic encryption using approximate computing techniques," in *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications, ICETE 2020 - Volume 2: SECRYPT, Lieusaint, Paris, France, July 8-10, 2020*, P. Samarati, S. D. C. di Vimercati, M. S. Obaidat, and J. Ben-Othman, Eds. ScitePress, 2020, pp. 380–387. [Online]. Available: https://doi.org/10.5220/0009828803800387

[21] 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

[22] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.

[23] 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.

[24] 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.

[25] 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

[26] 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

[27] 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.

[28] S. S. Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," in *Cryptographic Hardware and Embedded Systems - CHES*, 2015.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.
[34] 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.
[35] Xilinx, "Alveo U200 and U250 Data Center Accelerator Cards Data Sheet," https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.
[36] Xilinx, "Large FPGA Methodology Guide (UG872)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf.
[37] Xilinx, "Xilinx DMA IP Reference Drivers," https://github.com/Xilinx/dma_ip_drivers.
[38] 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.
[39] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," *arXiv preprint arXiv:2109.14704*, 2021.