Medha: *M*icrocoded *H*ardware *A*ccelerator for computing on *E*ncrypted *D*ata

Ahmet Can Mert[1], Aikata[1], Sunmin Kwon[2], Youngsam Shin[2], Donghoon Yoo[2], Yongwoo Lee[2] and Sujoy Sinha Roy[1]

[1] IAIK, Graz University of Technology, Austria,
`{sujoy.sinharoy,ahmet.mert,aikata}@iaik.tugraz.at`

[2] Samsung Advanced Institute of Technology, Suwon, Republic of Korea,
`{sunmin7.kwon,youngsam.shin,say.yoo,yw0803.lee}@samsung.com`

Abstract.
Homomorphic encryption enables computation on encrypted data, and hence it has a great potential in privacy-preserving outsourcing of computations to the cloud. Hardware acceleration of homomorphic encryption is crucial as software implementations are very slow. In this paper, we present design methodologies for building a programmable hardware accelerator for speeding up the cloud-side homomorphic evaluations on encrypted data.

First, we propose a divide-and-conquer technique that enables homomorphic evaluations in the polynomial ring $R_{Q,2N} = \mathbb{Z}_Q[x]/(x^{2N}+1)$ to use a hardware accelerator that has been built for the smaller ring $R_{Q,N} = \mathbb{Z}_Q[x]/(x^N+1)$. The technique makes it possible to use a single hardware accelerator flexibly for supporting several homomorphic encryption parameter sets.

Next, we present several architectural design methods that we use to realize the flexible and instruction-set accelerator architecture, which we call 'Medha'. 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 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.

Finally, we implement Medha in a Xilinx Alveo U250 FPGA and measure timing performances of the microcoded homomorphic addition, multiplication, key-switching, and rescaling routines for the leveled fully homomorphic encryption scheme RNS-HEAAN at 200 MHz clock frequency. For the large parameter sets $(\log Q, N) = (438, 2^{14})$ and $(546, 2^{15})$, Medha achieves accelerations by up to 68× and 78× times respectively compared to a highly optimized software implementation Microsoft SEAL running at 2.3 GHz.

Keywords: Homomorphic Encryption, CKKS, Flexible, Hardware Accelerator

1 Introduction

Cloud computing services are very popular and provide high-performance computational resources to the users [AFG+10]. 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) [RAD78] 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 [Gen09]. 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 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' that refreshes ciphertexts. This paper focuses on the hardware acceleration aspects of leveled-FHE. In the remaining part of the paper we will use the term HE to represent leveled-FHE. For evaluating arithmetic operations homomorphically, BFV [FV12] and BGV [BGV11] are popular. TFHE [CGGI20] is efficient for evaluating Boolean gates. For performing computations on encrypted *real numbers*, HEAAN [CKKS17] and its Residue Number System (RNS) variant RNS-HEAAN [CHK+18] are efficient. In fact, RNS-HEAAN is the fastest scheme for performing approximate computations on the encrypted *real* data.

Although a decade of research in algorithmic and mathematical optimizations has made HE 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 the plaintext. Therefore, the hardware acceleration of HE is crucial in reducing this performance gap. In the literature there are many works [RJV+15, TRG+20, RNH19, RTJ+20, FSK+21, XZH21] that implement selected building blocks of HE or present simulation-based performance results. A limitation of simulation-based works is that they do not fully capture the engineering challenges that appear only in actual hardware implementations, especially when the architectures are made very large to achieve high performance. Therefore, the simulation-based performances may change greatly when real hardware accelerators are made. Hence, developing HE hardware accelerator prototypes in real hardware is needed to accelerate HE in practice and at the same time identify potential research directions for performance improvements. Surprisingly, there are only a few works that report real HE accelerators [RJV+18, SRTJ+19, RLPD20, TRV20]. The FPGA-based programmable accelerators [SRTJ+19, TRV20] demonstrated latency reductions by one order compared to software implementations. Another FPGA-based accelerator 'HEAX' [RLPD20] obtained more than two orders of magnitude throughput with respect to software implementations. 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. On contrary to HEAX, the programmable accelerator [SRTJ+19] uses the same computational resources to execute several homomorphic evaluation routines. While programmability is a desired feature in accelerators, the one order speedup of HEAX [RLPD20] over the programmable processor [SRTJ+19, TRV20] may give us an impression that block-pipelined and specifically optimized accelerators are significantly superior to flexible accelerators for HE.

Homomorphic applications of different complexities (multiplicative depths) demand different parameter sets. Hence, supporting several parameter sets is another important yet currently unfulfilled requirement for the cloud-side accelerators. Almost all of the reported accelerators [RJV+18, SRTJ+19, RLPD20, TRV20] have been designed for specific parameter sets and they lack the flexibility to support more than one parameter set.

We address the above-mentioned research gaps by presenting a programmable and parameter-flexible hardware accelerator architecture 'Medha'. The architecture is implemented and verified in a Xilinx Alveo U250 Card. It accelerates the homomorphic addition, multiplication, key-switching, and re-scaling operations of RNS-HEAAN [CHK+18] for large parameter sets by around two orders of magnitude compared to software implementations.

Contributions: The main contributions of our paper reside at both algorithmic and architecture levels. First, we propose a design methodology that offers the flexibility to support several polynomial degrees using a fixed hardware accelerator. It efficiently performs homomorphic computations on 'large-degree' ciphertext-polynomials using compute units that have been optimized to handle 'small-degree' ciphertext-polynomials. Therefore, several homomorphic applications can be accelerated using the same architecture. Additionally, the proposed methodology reduces the on-chip memory and logic requirements.

On the architecture side, our main contributions are in the high-levels of the implementation hierarchy where different compute and memory elements are organized. To compute the arithmetic of residue polynomials, we design a novel Residue Polynomial Arithmetic Unit (RPAU) pragmatically. Our RPAU contains a multi-core Number Theoretic Transform (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 HE. 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 the largest supported polynomial degree with 2^{15} 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 HE subroutines for such large-degree polynomials.

At the highest level of the implementation hierarchy, several RPAUs are instantiated and interconnected. The parallel RPAUs must perform data exchanges between themselves during the modulus switching steps of the key-switching and rescaling operations. Trivially connecting every RPAU to the remaining RPAUs demands a quadratic number of nets and makes actual implementation infeasible when there are several large RPAUs in the architecture. Therefore, finding an optimal way of interconnecting the RPAUs is critical due to two main 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. Previous works, e.g., [RJV+15, RJV+18] bypassed that engineering problem by performing data exchanges happen via a shared off-chip memory at a great performance cost. Secondly, if the data transfer rate is compromised to reduce the number of interconnects, then there will be a drastic impact on the performance of key-switching and rescaling. After studying different ways of interconnecting the RPAUs, we propose a 'ring' styled interconnection with efficient scheduling of data transfers. The ring reduces the number of interconnects to a linear complexity without introducing any performance loss. We observe that the proposed ring is crucial for making large HE implementations feasible on SLR-based very large Xilinx FPGAs.

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 multi-core NTT-based 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.

Organization: The paper is organized as follows. Sec. 2 presents a brief background. Next, we present the proposed flexible design methodology for the polynomial degree and its applications to the RNS-HEAAN scheme in Sec. 3. In Sec. 4, the proposed flexible instruction-set accelerator is presented. The accelerator architecture is realized hierarchically starting from low-level polynomial arithmetic units in Sec. 4.2 and then organizing different compute and memory elements in Sec. 4.3. Detailed experimental results and comparisons are provided in Sec. 5 and the final section draws the conclusions.

2 Background

2.1 Notation

Let $\mathbb{Z}_Q$ represent the ring integers in the range $[0, Q-1]$. Any modular reduction by a modulus Q in $[0, Q-1]$ is denoted as $[.]_q$. The polynomial ring $R_{Q,N} = \mathbb{Z}_Q[x]/(x^N+1)$ contains polynomials of degree at most $N-1$ with coefficients in $\mathbb{Z}_Q$. An integer is represented using a normal font and lowercase letter, e.g., $a \in \mathbb{Z}_Q$. A polynomial is represented using a bold font and lowercase letter, $\boldsymbol{a} \in R_{Q,N}$. When a residue number system (RNS) is used with a composite modulus $Q = \prod_{i=0}^{L-1} q_i$, a polynomial in $R_{Q,N}$ becomes a vector of residue polynomials in the RNS. Let $\boldsymbol{a}[i] \in R_{q_i,N}$ be the i-th residue polynomial in the RNS representation of $\boldsymbol{a} \in R_{Q,N}$. $R^k_{Q,N}$ represents a k-tuple of polynomials from $R_{Q,N}$. Thereafter, the i-th residue polynomial of the j-th member from the tuple $\boldsymbol{a} \in R^k_{Q,N}$ is represented as $\boldsymbol{a}[j][i] \in R_{q_i,N}$.

The Number Theoretic Transform (NTT) of a polynomial $\boldsymbol{a}$ is represented by $\tilde{\boldsymbol{a}}$. The multiplication between two elements of a ring is denoted by the $\cdot$ operator. The coefficient-wise multiplication between two polynomials is denoted by the $\star$ operator. Multiplying all the coefficients of a polynomial $\boldsymbol{a}$ by an integer scalar c is denoted by $\boldsymbol{a} \odot c$ or $c \odot \boldsymbol{a}$.

2.2 Homomorphic Encryption

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 in the cloud. At the end of computations, the client receives the encrypted results from the cloud and performs decryptions locally to recover the plaintext results. The encryption and decryption operations are performed using the secret-key and private-key of the client, respectively.

Several ideal lattice-based homomorphic encryption schemes, e.g., BGV [BGV11], BFV [FV12], and HEAAN [CHK+18] use the following framework. Let, a client's secret-key be $\texttt{sk} = (1, \boldsymbol{s}) \in R^2_{Q,N}$ and the corresponding public-key be $\texttt{pk} = (\boldsymbol{b}, \boldsymbol{a}) \in R^2_{Q,N}$. Each key is a pair of polynomials in the polynomial ring $R_{Q,N}$ where Q is the coefficient-modulus and N is the polynomial ring degree. Client encrypts a message $\boldsymbol{m}$ using $\texttt{pk}$ and obtains the ciphertext $\texttt{ct} \leftarrow (\boldsymbol{c}_0 = \boldsymbol{r} \cdot \boldsymbol{b} + \boldsymbol{e}_0 + \boldsymbol{m}, \boldsymbol{c}_1 = \boldsymbol{r} \cdot \boldsymbol{a} + \boldsymbol{e}_1) \in R^2_{Q,N}$ where $\boldsymbol{e}_i$ is a Gaussian distributed error-polynomial and $\boldsymbol{r}$ is a uniformly random polynomial. Let, a cloud contains two ciphertexts $\texttt{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ and $\texttt{ct}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1) \in R^2_{Q,N}$ of the client as the encrypted messages $\boldsymbol{m}$ and $\boldsymbol{m}'$ respectively. The cloud can compute a valid encryption of $\boldsymbol{m} + \boldsymbol{m}'$ simply by adding the two ciphertexts as $\texttt{ct}_{\text{add}} \leftarrow (\boldsymbol{c}_0 + \boldsymbol{c}'_0, \boldsymbol{c}_1 + \boldsymbol{c}'_1) \in R^2_{Q,N}$. Computing an encryption of $\boldsymbol{m} \cdot \boldsymbol{m}'$ is relatively complex, scheme specific and involves several steps. First, the two ciphertexts are multiplied to obtain $\texttt{ct}_{\text{mult}} = (\boldsymbol{c}_0 \cdot \boldsymbol{c}'_0, \boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0, \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1) \in R^3_{Q,N}$. This intermediate result has three polynomial components and could be decrypted using $(1, \boldsymbol{s}, \boldsymbol{s}^2)$ but not using $\texttt{sk} = (1, \boldsymbol{s})$. Next, a special operation known as the 'Key-Switching', is used to transform the three-component ciphertext $\texttt{ct}_{\text{mult}}$, which is decryptable under $(1, \boldsymbol{s}, \boldsymbol{s}^2)$, into the two-component ciphertext $\texttt{ct}_{\text{relin}}$ decryptable under $(1, \boldsymbol{s})$. In this context, key-switching is called re-linearization as it produces a linear ciphertext.

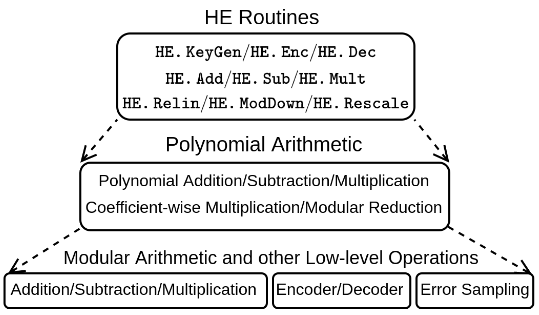
HE Routines
HE. KeyGen/HE. Enc/HE. Dec
HE. Add/HE. Sub/HE. Mult
HE. Relin/HE. ModDown/HE. Rescale
Polynomial Arithmetic
Polynomial Addition/Subtraction/Multiplication
Coefficient-wise Multiplication/Modular Reduction
Modular Arithmetic and other Low-level Operations
Addition/Subtraction/Multiplication
Encoder/Decoder
Error Sampling

Figure 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 error sampling at the lowest level. Homomorphic evaluations on ciphertexts using homomorphic addition, subtraction, multiplication, relinearization, 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.

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.

2.3 Residue Number System (RNS)

In HE, typically the size of Q is several hundreds to several thousands of bits. Arithmetic operations with a large coefficient modulus require multi-precision arithmetic and it reduces the performance of HE-based applications. The Residue Number System (RNS) is a popular choice for implementing HE on hardware and software platforms as it enables the parallelization of computations. To work with the RNS, the modulus Q is chosen to be a product of small coprimes q_i such that $Q = \prod_{i=0}^{L-1} q_i$. These coprimes form the base $\mathcal{B} = \{q_0, \ldots, q_{L-1}\}$ of the RNS. Any long integer $a \in \mathbb{Z}_Q$ gets mapped into the set of small residues $a_i = a \pmod{q_i}$ in the RNS. On the other hand, a set of small residues a_i in the RNS can be combined together using the Chinese Remainder Theorem (CRT) to obtain the integer $a \in \mathbb{Z}_Q$.

In ideal lattice-based homomorphic encryption schemes, with the application of RNS, a polynomial $\boldsymbol{a} \in R_{Q,N}$ gets mapped into the set of residue polynomials in the RNS base. These residue polynomials have small coefficients and therefore they can be processed without requiring expensive multi-precision arithmetic operations. Additionally, on platforms with parallelism, e.g., hardware platforms or superscalar processors, a set of residue polynomials can be processed in parallel. In summary, the application of RNS makes implementations of homomorphic encryption simpler, modular, and faster.

2.4 RNS-HEAAN

RNS-HEAAN [CHK+18] is an RNS variant of the original HEAAN [CKKS17] homomorphic encryption scheme. The original HEAAN [CKKS17] scheme uses a large modulus (which is a power of two) and its expensive multi-precision integer arithmetic becomes a performance bottleneck. Our hardware accelerator Medha has been optimized to accelerate the cloud-side homomorphic operations of RNS-HEAAN. In this section, we briefly describe the cloud side arithmetic routines of the RNS-HEAAN variant that is used in SEAL [SEA20]. We chose SEAL's RNS-HEAAN variant for our implementation to make fair performance comparisons with SEAL [SEA20] and HEAX [RLPD20].

To use RNS-HEAAN in an application, the first step is to set up the scheme parameters such as the degree N of the polynomial ring, size of the maximum ciphertext modulus Q_L, RNS-base $\mathcal{B}_L$ with L small prime moduli, etc., depending on the maximum multiplicative depth L required by the application for a desired level of security. The multiplicative depth is the maximum number of consecutive homomorphic multiplications that can be performed before a wrong result is produced due to too much noise. After the setup phase, the key generation phase generates the public-private key pair and the relinearization key. The relinearization key which is a key-switching key, say $\mathtt{KSK} = (\mathtt{KSK}_0, \mathtt{KSK}_1)$, is sent to the cloud as it will be used to relinearize the result of a homomorphic multiplication using the key-switching procedure. Each of $\mathtt{KSK}_0$ and $\mathtt{KSK}_1$ is an L-tuple and resides in $R^L_{pQ_L,N}$. Here p is a special prime that is used only during the key-switching operation. Sometimes we denote this special prime by q_L and consider the extended RNS base to be $\mathcal{B}_L = \{q_0, \ldots, q_{L-1}, p\}$. We use the notation $\mathtt{KSK}_0[i]$ to select the i-th element from the L-tuple. This element is in $R_{pQ_L,N}$ and therefore it consists of $(L+1)$ residue polynomials in the RNS representation. We use the notation $\mathtt{KSK}_0[i][j]$ to represent the j-th residue polynomial of $\mathtt{KSK}_0[i]$. In summary, each of $\mathtt{KSK}_0$ and $\mathtt{KSK}_1$ is a vector of $L(L+1)$ residue polynomials. A ciphertext at the multiplicative level $l \leq L$ is represented using the RNS base $\mathcal{B}_l = \{q_0, \ldots, q_{l-1}\}$.

To get a detailed description of RNS-HEAAN, the readers may follow the original publication [CHK+18] and SEAL's [SEA20] RNS-HEAAN implementation documentation.

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

- $\mathtt{HEAAN.Add}(\mathtt{ct}, \mathtt{ct}')$: It adds the respective polynomials of the two ciphertexts $\mathtt{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R^2_{Q_l,N}$ and $\mathtt{ct}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1) \in R^2_{Q_l,N}$, and computes $\mathtt{ct}_{\text{add}} = (\boldsymbol{d}_0, \boldsymbol{d}_1)$ where $\boldsymbol{d}_0 = \boldsymbol{c}_0 + \boldsymbol{c}'_0 \in R^2_{Q_l,N}$ and $\boldsymbol{d}_1 = \boldsymbol{c}_1 + \boldsymbol{c}'_1 \in R^2_{Q_l,N}$.

- $\mathtt{HEAAN.Mult}(\mathtt{ct}, \mathtt{ct}')$: It multiplies two input ciphertexts $\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R^2_{Q_l,N}$ and $\mathtt{ct}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1) \in R^2_{Q_l,N}$, and computes $\boldsymbol{d}_0 = \boldsymbol{c}_0 \cdot \boldsymbol{c}'_0 \in R_{Q_l,N}$, $\boldsymbol{d}_1 = \boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0 \in R_{Q_l,N}$, and $\boldsymbol{d}_2 = \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1 \in R_{Q_l,N}$. The output is the non-linear ciphertext $\mathtt{dt} = (\boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2) \in R^3_{Q_l,N}$.

- $\mathtt{HEAAN.Relin}(\mathtt{dt}, \mathtt{KSK})$: It re-linearizes the result of $\mathtt{HEAAN.Mult}$ and produces a ciphertext with two polynomial components so that it is decryptable under the secret key. Let $\boldsymbol{d}'_2[i] = \left[\boldsymbol{d}_2\right]_{q_i}$ for $0 \leq i < l$. Now compute $\mathtt{ct}'' = (\boldsymbol{c}''_0, \boldsymbol{c}''_1)$ where $\boldsymbol{c}''_0 = \sum_{i=0}^{l-1} \boldsymbol{d}'_2[i] \cdot \mathtt{KSK}_0[i] \in R_{pQ_l,N}$ and $\boldsymbol{c}''_1 = \sum_{i=0}^{l-1} \boldsymbol{d}'_2[i] \cdot \mathtt{KSK}_1[i] \in R_{pQ_l,N}$. Finally, output the relinearized ciphertext $\mathtt{ct}_{\text{relin}} = (\boldsymbol{d}_0, \boldsymbol{d}_1) + (\mathtt{HEAAN.ModDown}(\boldsymbol{c}''_0), \mathtt{HEAAN.ModDown}(\boldsymbol{c}''_1)) \pmod{Q_l}$. The $\mathtt{HEAAN.ModDown}$ operation is used to reduce coefficient modulus from pQ_l to Q_l and is computationally similar to the rescaling operation.

- $\mathtt{HEAAN.Rescale}(\boldsymbol{c})$: It takes a ciphertext-polynomial $\boldsymbol{c} \in R_{Q_l,N}$ with level l and produces a ciphertext element with level $l-1$. Let $\boldsymbol{c}' \in R_{Q_{l-1},N}$ such that $\boldsymbol{c}'[i] = \boldsymbol{c}[l] \pmod{q_i}$ for $0 \leq i \leq l-1$. Then, compute $\boldsymbol{c}'' = \boldsymbol{c} - \boldsymbol{c}' \in R_{Q_{l-1},N}$. Finally, output the rescaled ciphertext element $\boldsymbol{c}'' = q_l^{-1} \odot \boldsymbol{c}'' \in R_{Q_{l-1},N}$. To rescale a ciphertext that consists of two polynomials, the above procedure is applied to both the polynomials.

Algorithm 1 HEAAN.Add Algorithm

```
In: $\mathtt{ct} = (\tilde{c}_0, \tilde{c}_1)$, $\mathtt{ct}' = (\tilde{c}'_0, \tilde{c}'_1) \in R^2_{Q_l,N}$
Out: $\mathtt{d} = (\tilde{d}_0, \tilde{d}_1) \in R^2_{Q_l,N}$
1: $\tilde{d}_0 \leftarrow \tilde{c}_0 + \tilde{c}'_0$
2: $\tilde{d}_1 \leftarrow \tilde{c}_1 + \tilde{c}'_1$
```

Algorithm 2 HEAAN.Mult Algorithm

```
In: $\mathtt{ct} = (\tilde{c}_0, \tilde{c}_1)$, $\mathtt{ct}' = (\tilde{c}'_0, \tilde{c}'_1) \in R^2_{Q_l,N}$
Out: $\mathtt{d} = (\tilde{d}_0, \tilde{d}_1, \tilde{d}_2) \in R^3_{Q_l,N}$
1: $\tilde{d}_0 \leftarrow \tilde{c}_0 \star \tilde{c}'_0$, $\tilde{d}_2 \leftarrow \tilde{c}_1 \star \tilde{c}'_1$
2: $\tilde{d}_1 \leftarrow \tilde{c}_0 \star \tilde{c}'_1 + \tilde{c}_1 \star \tilde{c}'_0$
```

Algorithm 3 HEAAN.ModDown Algorithm

```
In: $\tilde{d} \in R_{pQ_l,N}$
Out: $\tilde{d}' \in R_{Q_l,N}$
1: $t \leftarrow \mathtt{INTT}(\tilde{d}[l])$
2: for $i = 0$ to $l - 1$ do
3:     $\tilde{t} \leftarrow \mathtt{NTT}([t]_{q_i})$                ▷ in $\mathbb{Z}_{q_i}$
4:     $\tilde{d}'[i] \leftarrow [p^{-1} \odot (\tilde{d}[i] - \tilde{t})]_{q_i}$
5: end for
```

Algorithm 4 HEAAN.Rescale Algorithm

```
In: $\tilde{d} \in R_{Q_l,N}$
Out: $\tilde{d}' \in R_{Q_{l-1},N}$
1: $t \leftarrow \mathtt{INTT}(\tilde{d}[l-1])$
2: for $i = 0$ to $l - 2$ do
3:     $\tilde{t} \leftarrow \mathtt{NTT}([t]_{q_i})$                ▷ in $\mathbb{Z}_{q_i}$
4:     $\tilde{d}'[i] \leftarrow [q_l^{-1} \odot (\tilde{d}[i] - \tilde{t})]_{q_i}$
5: end for
```

Algorithm 5 HEAAN.Relin Algorithm

```
In: $\mathtt{d} = (\tilde{d}_0, \tilde{d}_1, \tilde{d}_2) \in R^3_{Q_l,N}$, $\tilde{\mathtt{KSK}}_0 \in R^l_{pQ_l,N}$, $\tilde{\mathtt{KSK}}_1 \in R^l_{pQ_l,N}$
Out: $\mathtt{d}' = (\tilde{d}'_0, \tilde{d}'_1) \in R^2_{Q_l,N}$
1: for $j = 0$ to $l - 1$ do
2:     $d_2[j] \leftarrow \mathtt{INTT}(\tilde{d}_2[j])$                ▷ in $\mathbb{Z}_{q_j}$
3: end for
4: for $j = 0$ to $l$ do                ▷ Here $q_l$ is used to represent special prime $p$
5:     $(\tilde{c}''_0[j], \tilde{c}''_1[j]) \leftarrow 0$
6:     for $i = 0$ to $l - 1$ do
7:         $\tilde{r} \leftarrow \mathtt{NTT}([d_2[i]]_{q_j})$                ▷ in $\mathbb{Z}_{q_j}$
8:         $\tilde{c}''_0[j] \leftarrow [\tilde{c}''_0[j] + \tilde{\mathtt{KSK}}_0[i][j] \star \tilde{r}]_{q_j}$, $\tilde{c}''_1[j] \leftarrow [\tilde{c}''_1[j] + \tilde{\mathtt{KSK}}_1[i][j] \star \tilde{r}]_{q_j}$
9:     end for
10: end for
11: $\tilde{d}'_0 \leftarrow \tilde{d}_0 + \mathtt{HEAAN.ModDown}(\tilde{c}''_0)$, $\tilde{d}'_1 \leftarrow \tilde{d}_1 + \mathtt{HEAAN.ModDown}(\tilde{c}''_1)$
```

In optimized HEAAN implementations, the ciphertexts and key-switching key are kept in the NTT domain with RNS bases, which improves the performance. The NTT is a method used for efficient implementation of large-degree polynomial multiplication operation, which is detailed in Sec. 2.5. The textbook HEAAN subroutines above were presented using non-NTT representation for the sake of simplicity. The steps of the optimized homomorphic HEAAN.Add, HEAAN.Mult, HEAAN.Relin, HEAAN.ModDown and HEAAN.Rescale operations using NTT domain polynomials are given in Algorithm 1, 2, 5, 3 and 4 respectively. The tilde symbol on top of a variable indicates that the polynomial is in the NTT domain. From now on, we use the optimized subroutines with NTT representation.

2.5 Number Theoretic Transform (NTT)

The multiplication of very large degree polynomials is one of the major performance bottlenecks for the HE implementations. The Number Theoretic Transform or NTT enables fast polynomial multiplications by reducing the complexity of polynomial multiplication to $\mathcal{O}(n \cdot \log n)$, and it is extensively employed for the implementation of HE schemes. The NTT is defined as the Discrete Fourier Transform over $\mathbb{Z}_q$. An N-point NTT operation transforms a polynomial $\boldsymbol{a}$ of degree $N-1$ degree polynomial into another $N-1$ degree polynomial $\tilde{\boldsymbol{a}}$. The NTT uses the powers of N-th root of unity ω (also referred to as twiddle factors) which satisfies $\omega^n \equiv 1 \pmod q$ and $\omega^i \neq 1 \pmod q$ $\forall i < N$, where $q \equiv 1 \pmod N$. Similarly, inverse NTT (INTT) follows the same method with the modular inverse of ω and the resulting coefficients should be scaled by $1/N$.

When the polynomial ring is in the form of $\mathbb{Z}_q[x]/x^N + 1$ and $q \equiv 1 \pmod{2N}$, *negative wrapped convolution* technique enables an efficient NTT-based polynomial multiplication method. First, NTT is performed on input polynomials, then the resulting polynomials are coefficient-wise multiplied and finally, the INTT operation is performed. However, this technique requires input and output polynomials to be pre- and post-processed with $2N$-th root of unity, respectively. Roy *et al.* [SRVM$^+$14] and Poppelmann *et al.* [POG15] showed how to merge pre-processing with NTT and post-processing with INTT, respectively. These new NTT and INTT operations use $2N$-th root of unity and its modular inverse, respectively. Polynomial multiplication operation $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \pmod{x^N + 1}$ can be performed using new NTT and INTT operations as shown in Eqn. 1.

$$\boldsymbol{c} = \texttt{INTT}(\texttt{NTT}(\boldsymbol{a}) \star \texttt{NTT}(\boldsymbol{b})) \tag{1}$$

3 Flexible Design Methodology

3.1 Homomorphic Encryption Applications Require Flexibility

Different homomorphic applications require different multiplicative depths, hence the parameter sets of the underlying homomorphic encryption instances change with the complexity of the application. For example, the SEAL [SEA20] software library uses RNS-HEAAN parameters $(\log_2 Q, N) = (109, 2^{12})$, $(218, 2^{13})$ and $(438, 2^{14})$ for the multiplicative depths 2, 4, and 8, respectively. As software implementations are inherently flexible, such changes in the parameter sets do not require a re-implementation of the software – they affect only the performance and memory allocation overheads. E.g., SEAL [SEA20] handles the memory allocation for different parameters flexibly by dynamically allocating memory blocks in the heap. On the other hand, hardware implementations are generally not flexible as the compute cores and memory elements are *hardwired* by a fixed set of wires. As a consequence, when the hardware acceleration of cloud-side homomorphic evaluation routines is considered, the drastic changes in the sizes of the parameter sets pose a major challenge. To satisfy the flexibility requirements associated with different applications, one naive approach will be to install several parameter-specific optimized accelerators in the cloud. With that, the cloud can choose an appropriate accelerator for evaluating a given homomorphic application. This approach will give optimal execution time for any application, but will cause redundancy by requiring several accelerator units. A second approach will be to use a single accelerator which has been designed for a large parameter. With this approach, all homomorphic applications irrespective of their complexities will have to use the large and fixed parameter set. Therefore, this approach will cause poor performances for applications that do not require a large parameter set. In this paper, we propose a third approach which is a divide-and-conquer approach. It enables using a single hardware accelerator, that has been designed for a smaller parameter, for supporting several larger parameter sets flexibly without causing any major performance loss.

For the cloud-side operations, the size of coefficient modulus Q and the polynomial degree N change with the multiplicative depth. Attaining flexibility in the modulus size is relatively easy if the homomorphic encryption scheme uses RNS. As explained in Sec. 2.4, an RNS-based homomorphic encryption scheme operates using a set of moduli. Therefore, a change in the size of the coefficient modulus is equivalent to addition or deletion of moduli to/from the RNS base. The hardware accelerators [SRTJ$^+$19, TRV20] implement Q-flexibility by instantiating several parallel residue arithmetic processors in the architecture. Each such processor is used in a time-shared manner to support more than one moduli.

Implementing flexibility for the polynomial degree N in a hardware accelerator is not straightforward. In the literature, there is only one accelerator F1 [FSK+21] that supports multiple parameter sets with the polynomial degrees ranging from 2^{10} to 2^{14} by allocating computational resources for the largest parameter. For smaller parameters, it uses a subset of the available computational resources. This approach increases the complexity of the implementation as it requires the computational resources for the largest parameter. Also, extra logic elements are required to bypass a subset of idle computational resources for smaller parameters.

Our proposed flexible accelerator design methodology is fundamentally different from the method used in F1. It uses arithmetic units that have been optimized for a fixed-and-smaller parameter to perform operations for larger parameters.

3.2 Design Methodology for Flexibility in Polynomial Degree

Let's assume that there is an accelerator architecture that has been optimized for the homomorphic operations with polynomial degree N. The research problem is: How could we use the accelerator for larger parameter sets, e.g., with the polynomial degrees $2N$, $4N$, etc.? To solve the problem, we propose a divide-and-conquer approach. We apply the polynomial version of CRT to the ciphertexts (which consist of residue polynomials) and split a ciphertext with a large polynomial degree (i.e., $\texttt{ct} \in R^2_{Q,2N}$) into several ciphertexts with smaller polynomial degrees (i.e., $\texttt{ct}' \in R^4_{Q,N}$). Next, these smaller ciphertexts (with polynomial degrees N) are processed using the given hardware accelerator that has been optimized for the polynomial degree N. The splitting can be applied recursively.

Let the irreducible polynomial $\boldsymbol{f}$ of the ring $R_{Q,2N} = \mathbb{Z}_Q[x]/\boldsymbol{f}$ be of the form $\boldsymbol{f} = x^{2N} + 1$. Let all the moduli q_i in the RNS base of the Q be primes such that $q_i \equiv 1 \pmod{4N}$. Then $4N$-th primitive root of the unity will exist in $\mathbb{Z}_{q_i}$. Let ζ_{4N} be one such primitive root of the unity, i.e., $\zeta_{4N}^{4N} \equiv 1 \pmod{q_i}$ and $\zeta_{4N}^{2N} \equiv -1 \pmod{q_i}$. Then, $\boldsymbol{f}$ can be re-written as $x^{2N} - \zeta_{4N}^{2N}$ and factored into polynomials $\boldsymbol{f}_p$ and $\boldsymbol{f}_m$ in $\mathbb{Z}_Q$ as shown in Eqn. 2.

$$\boldsymbol{f}(x) = \boldsymbol{f}_p(x) \cdot \boldsymbol{f}_m(x) = (x^N - \zeta_{4N}^N) \cdot (x^N + \zeta_{4N}^N) \tag{2}$$

Note that $\boldsymbol{f}$ is irreducible over integers but not over $\mathbb{Z}_Q$. Since $\boldsymbol{f}_p$ and $\boldsymbol{f}_m$ are coprime, the isomorphism $R_{q_i,2N} = \mathbb{Z}_{q_i}[x]/\boldsymbol{f} \mapsto \mathbb{Z}_{q_i}[x]/\boldsymbol{f}_p \times \mathbb{Z}_{q_i}[x]/\boldsymbol{f}_m$ exists. Thus, any operation in the polynomial ring $R_{q_i,2N}$ can be mapped to smaller operations in the polynomial rings $\mathbb{Z}_{q_i}[x]/\boldsymbol{f}_p$ and $\mathbb{Z}_{q_i}[x]/\boldsymbol{f}_m$. From now on, we use $R_{q_i,N,p}$ and $R_{q_i,N,m}$ (*cf.* $R_{Q,N,p}$ and $R_{Q,N,m}$) to represent the polynomial rings $\mathbb{Z}_{q_i}[x]/\boldsymbol{f}_p$ and $\mathbb{Z}_{q_i}[x]/\boldsymbol{f}_m$ (*cf.* $\mathbb{Z}_Q[x]/\boldsymbol{f}_p$ and $\mathbb{Z}_Q[x]/\boldsymbol{f}_m$), respectively.

Using the isomorphism a large ciphertext $\texttt{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R^2_{Q,2N}$ can be split into two smaller ciphertexts $\texttt{ct}^{(p)} = (\boldsymbol{c}_0^{(p)}, \boldsymbol{c}_1^{(p)}) \in R^2_{Q,N,p}$ and $\texttt{ct}^{(m)} = (\boldsymbol{c}_0^{(m)}, \boldsymbol{c}_1^{(m)}) \in R^2_{Q,N,m}$. After that, the homomorphic operations can be performed on the smaller ciphertexts. This approach enables efficient re-use of a fixed hardware architecture that has been optimized for the smaller parameter with the polynomial degree N. Additionally, management of the smaller data variables in the limited on-chip memory of a hardware accelerator becomes relatively easier.

The splitting of a ciphertext is essentially the splittings of its polynomials. A polynomial $\boldsymbol{a} \in R_{Q,2N}$ is split into the two polynomials $\boldsymbol{a}_p = \boldsymbol{a} \pmod{\boldsymbol{f}_p} \in R_{Q,N,p}$ and $\boldsymbol{a}_m = \boldsymbol{a} \pmod{\boldsymbol{f}_m} \in R_{Q,N,m}$. On the opposite mapping, the two smaller polynomials $\boldsymbol{a}_p$ and $\boldsymbol{a}_m$ can be joined using the CRT to obtain the large polynomial $\boldsymbol{a}$ as

$$\boldsymbol{a} = (\boldsymbol{a}_p + \boldsymbol{a}_m)\frac{1}{2} + (\boldsymbol{a}_p - \boldsymbol{a}_m)\frac{\zeta_{4N}^{-N}}{2}x^N \pmod{Q}. \tag{3}$$

Algorithm 6 `HEAAN.Add` Algorithm for $2N$

```
In: $\mathtt{ct}^{(p)} = (\tilde{\boldsymbol{c}}_0^{(p)}, \tilde{\boldsymbol{c}}_1^{(p)}) \in R^2_{Q_l,N,p}$
In: $\mathtt{ct}^{(m)} = (\tilde{\boldsymbol{c}}_0^{(m)}, \tilde{\boldsymbol{c}}_1^{(m)}) \in R^2_{Q_l,N,m}$
In: $\mathtt{ct}'^{(p)} = (\tilde{\boldsymbol{c}}_0'^{(p)}, \tilde{\boldsymbol{c}}_1'^{(p)}) \in R^2_{Q_l,N,p}$
In: $\mathtt{ct}'^{(m)} = (\tilde{\boldsymbol{c}}_0'^{(m)}, \tilde{\boldsymbol{c}}_1'^{(m)}) \in R^2_{Q_l,N,m}$
Out: $\mathtt{d}^{(p)} = (\tilde{\boldsymbol{d}}_0^{(p)}, \tilde{\boldsymbol{d}}_1^{(p)}) \in R^2_{Q_l,N,p}$
Out: $\mathtt{d}^{(m)} = (\tilde{\boldsymbol{d}}_0^{(m)}, \tilde{\boldsymbol{d}}_1^{(m)}) \in R^2_{Q_l,N,m}$
1: $\tilde{\boldsymbol{d}}_0^{(p)} \leftarrow \tilde{\boldsymbol{c}}_0^{(p)} + \tilde{\boldsymbol{c}}_0'^{(p)}$
2: $\tilde{\boldsymbol{d}}_1^{(p)} \leftarrow \tilde{\boldsymbol{c}}_1^{(p)} + \tilde{\boldsymbol{c}}_1'^{(p)}$
3: $\tilde{\boldsymbol{d}}_0^{(m)} \leftarrow \tilde{\boldsymbol{c}}_0^{(m)} + \tilde{\boldsymbol{c}}_0'^{(m)}$
4: $\tilde{\boldsymbol{d}}_1^{(m)} \leftarrow \tilde{\boldsymbol{c}}_1^{(m)} + \tilde{\boldsymbol{c}}_1'^{(m)}$
```

Algorithm 7 `HEAAN.Mult` Algorithm for $2N$

```
In: $\mathtt{ct}^{(p)} = (\tilde{\boldsymbol{c}}_0^{(p)}, \tilde{\boldsymbol{c}}_1^{(p)}) \in R^2_{Q_l,N,p}$
In: $\mathtt{ct}^{(m)} = (\tilde{\boldsymbol{c}}_0^{(m)}, \tilde{\boldsymbol{c}}_1^{(m)}) \in R^2_{Q_l,N,m}$
In: $\mathtt{ct}'^{(p)} = (\tilde{\boldsymbol{c}}_0'^{(p)}, \tilde{\boldsymbol{c}}_1'^{(p)}) \in R^2_{Q_l,N,p}$
In: $\mathtt{ct}'^{(m)} = (\tilde{\boldsymbol{c}}_0'^{(m)}, \tilde{\boldsymbol{c}}_1'^{(m)}) \in R^2_{Q_l,N,m}$
Out: $\mathtt{d}^{(p)} = (\tilde{\boldsymbol{d}}_0^{(p)}, \tilde{\boldsymbol{d}}_1^{(p)}, \tilde{\boldsymbol{d}}_2^{(p)}) \in R^3_{Q_l,N,p}$
Out: $\mathtt{d}^{(m)} = (\tilde{\boldsymbol{d}}_0^{(m)}, \tilde{\boldsymbol{d}}_1^{(m)}, \tilde{\boldsymbol{d}}_2^{(m)}) \in R^3_{Q_l,N,m}$
1: $\tilde{\boldsymbol{d}}_0^{(p)} \leftarrow \tilde{\boldsymbol{c}}_0^{(p)} \star \tilde{\boldsymbol{c}}_0'^{(p)}$, $\tilde{\boldsymbol{d}}_2^{(p)} \leftarrow \tilde{\boldsymbol{c}}_1^{(p)} \star \tilde{\boldsymbol{c}}_1'^{(p)}$
2: $\tilde{\boldsymbol{d}}_1^{(p)} \leftarrow \tilde{\boldsymbol{c}}_0^{(p)} \star \tilde{\boldsymbol{c}}_1'^{(p)} + \tilde{\boldsymbol{c}}_1^{(p)} \star \tilde{\boldsymbol{c}}_0'^{(p)}$
3: $\tilde{\boldsymbol{d}}_0^{(m)} \leftarrow \tilde{\boldsymbol{c}}_0^{(m)} \star \tilde{\boldsymbol{c}}_0'^{(m)}$, $\tilde{\boldsymbol{d}}_2^{(m)} \leftarrow \tilde{\boldsymbol{c}}_1^{(m)} \star \tilde{\boldsymbol{c}}_1'^{(m)}$
4: $\tilde{\boldsymbol{d}}_1^{(m)} \leftarrow \tilde{\boldsymbol{c}}_0^{(m)} \star \tilde{\boldsymbol{c}}_1'^{(m)} + \tilde{\boldsymbol{c}}_1^{(m)} \star \tilde{\boldsymbol{c}}_0'^{(m)}$
```

The same applies to the key-switching key. The splitting and joining operations each have a linear time complexity. They can be performed in the software or in the hardware. For example, a software system that is using a hardware accelerator as a coprocessor, can do the splitting job before sending the small ciphertexts to the hardware. Similarly, the software system can combine the partial results returned by the hardware. In the remaining part of the paper, we will use the two functions `Split`() and `Join`() to perform the splitting and joining operations respectively.

3.3 RNS-HEAAN Subroutines with Flexible Method

Here we provide concrete algorithmic descriptions of the subroutines of RNS-HEAAN to implement the flexibility that we discussed in the previous section.

`HEAAN.Add` and `HEAAN.Mult` subroutines: Let two large ciphertexts be `ct` and $\mathtt{ct}' \in R^2_{Q_l,2N}$. Using the `Split`() function, each of them is split into two smaller ciphertexts. Algorithm 6 and Algorithm 7 show how the smaller ciphertexts are processed to perform the homomorphic addition and multiplication in the larger ring $R_{Q_l,2N}$. In the RNS these polynomial operations are performed on the residue polynomials.

`HEAAN.Relin` and `HEAAN.Rescale` subroutines: The steps of relinearization and rescaling operations for the polynomial degree of $2N$ are shown in Algorithm 8 and Algorithm 9, respectively. The mod-down and rescaling operations are very similar as shown in Algorithm 3 and Algorithm 4, thus the steps of the mod-down operation for $2N$ can easily be constructed from the steps of rescaling operation (Algorithm 9). The relinearization, mod-down and rescaling operations for the polynomial degree of $2N$ are more complex than the homomorphic addition and multiplication operations as they require the `Split`/`Join` and `NTT`/`INTT` operations. As shown in step 7 of Algorithm 5, the key switching multiplication part of the relinearization operation requires the polynomials of the ciphertext $\boldsymbol{d}_2$ to be reduced by q_j, which cannot be performed independently on $\boldsymbol{d}_2^{(p)}$ and $\boldsymbol{d}_2^{(m)}$. Therefore, before the modular reduction by q_j the two small polynomials are joined using `Join` in step 3 of Algorithm 8. After the modular reduction, the reduced large polynomial is again split into two small polynomials using `Split` in step 8 of Algorithm 8. Similarly, the mod-down and rescaling operations also require the split and join operations as they involve modular reductions.

Note that the resultant polynomials after a `Split` operation reside in two different

Algorithm 8 HEAAN.Relin Algorithm for $2N$

```
In: $\mathtt{d}^{(p)} = (\tilde{\boldsymbol{d}}_0^{(p)}, \tilde{\boldsymbol{d}}_1^{(p)}, \tilde{\boldsymbol{d}}_2^{(p)}) \in R^3_{Q_l,N,p}$, $\mathtt{d}^{(m)} = (\tilde{\boldsymbol{d}}_0^{(m)}, \tilde{\boldsymbol{d}}_1^{(m)}, \tilde{\boldsymbol{d}}_2^{(m)}) \in R^3_{Q_l,N,m}$
In: $\mathtt{K\tilde{S}K}_0^{(p)} \in R^l_{pQ_l,N,p}$, $\mathtt{K\tilde{S}K}_1^{(p)} \in R^l_{pQ_l,N,p}$, $\mathtt{K\tilde{S}K}_0^{(m)} \in R^l_{pQ_l,N,m}$, $\mathtt{K\tilde{S}K}_1^{(m)} \in R^l_{pQ_l,N,m}$
Out: $\mathtt{d}^{(p)} = (\tilde{\boldsymbol{d}}_0^{(p)}, \tilde{\boldsymbol{d}}_1^{(p)}) \in R^2_{Q_l,N,p}$, $\mathtt{d}^{(m)} = (\tilde{\boldsymbol{d}}_0^{(m)}, \tilde{\boldsymbol{d}}_1^{(m)}) \in R^2_{Q_l,N,m}$
 1: for $j = 0$ to $l - 1$ do
 2:     $\boldsymbol{d}_2^{(p)}[j] \leftarrow \mathtt{INTT_p}(\tilde{\boldsymbol{d}}_2^{(p)}[j])$, $\boldsymbol{d}_2^{(m)}[j] \leftarrow \mathtt{INTT_m}(\tilde{\boldsymbol{d}}_2^{(m)}[j])$    ▷ in $\mathbb{Z}_{q_j}$
 3:     $\boldsymbol{d}_2[j] \leftarrow \mathtt{Join}(\boldsymbol{d}_2^{(p)}[j], \boldsymbol{d}_2^{(m)}[j])$
 4: end for
 5: for $j = 0$ to $l$ do    ▷ Here $q_l$ is used to represent $p$
 6:     $(\boldsymbol{c}_0''^{(p)}[j], \boldsymbol{c}_1''^{(p)}[j], \boldsymbol{c}_0''^{(m)}[j], \boldsymbol{c}_1''^{(m)}[j]) \leftarrow 0$
 7:     for $i = 0$ to $l - 1$ do
 8:         $\boldsymbol{r}^{(p)}, \boldsymbol{r}^{(m)} \leftarrow \mathtt{Split}([\boldsymbol{d}_2[i]]_{q_j})$
 9:         $\tilde{\boldsymbol{r}}^{(p)} \leftarrow \mathtt{NTT_p}(\boldsymbol{r}^{(p)})$, $\tilde{\boldsymbol{r}}^{(m)} \leftarrow \mathtt{NTT_m}(\boldsymbol{r}^{(m)})$    ▷ in $\mathbb{Z}_{q_j}$
10:         $\boldsymbol{c}_0''^{(p)}[j] \leftarrow [\boldsymbol{c}_0''^{(p)}[j] + \mathtt{K\tilde{S}K}_0^{(p)}[i][j] \star \tilde{\boldsymbol{r}}^{(p)}]_{q_j}$
11:         $\boldsymbol{c}_1''^{(p)}[j] \leftarrow [\boldsymbol{c}_1''^{(p)}[j] + \mathtt{K\tilde{S}K}_1^{(p)}[i][j] \star \tilde{\boldsymbol{r}}^{(p)}]_{q_j}$
12:         $\boldsymbol{c}_0''^{(m)}[j] \leftarrow [\boldsymbol{c}_0''^{(m)}[j] + \mathtt{K\tilde{S}K}_0^{(m)}[i][j] \star \tilde{\boldsymbol{r}}^{(m)}]_{q_j}$
13:         $\boldsymbol{c}_1''^{(m)}[j] \leftarrow [\boldsymbol{c}_1''^{(m)}[j] + \mathtt{K\tilde{S}K}_1^{(m)}[i][j] \star \tilde{\boldsymbol{r}}^{(m)}]_{q_j}$
14:     end for
15: end for
16: $(\tilde{\boldsymbol{d}}_0'^{(p)}, \tilde{\boldsymbol{d}}_0'^{(m)}) \leftarrow (\tilde{\boldsymbol{d}}_0^{(p)}, \tilde{\boldsymbol{d}}_0^{(m)}) + (\mathtt{HEAAN.ModDown}(\tilde{\boldsymbol{c}}_0''^{(p)}), \mathtt{HEAAN.ModDown}(\tilde{\boldsymbol{c}}_0''^{(m)}))$
17: $(\tilde{\boldsymbol{d}}_1'^{(p)}, \tilde{\boldsymbol{d}}_1'^{(m)}) \leftarrow (\tilde{\boldsymbol{d}}_1^{(p)}, \tilde{\boldsymbol{d}}_1^{(m)}) + (\mathtt{HEAAN.ModDown}(\tilde{\boldsymbol{c}}_1''^{(p)}), \mathtt{HEAAN.ModDown}(\tilde{\boldsymbol{c}}_1''^{(m)}))$
```

Algorithm 9 HEAAN.Rescale Algorithm for $2N$

```
In: $\tilde{\boldsymbol{d}}^{(p)} \in R_{Q_l,N,p}, \tilde{\boldsymbol{d}}^{(m)} \in R_{Q_l,N,m}$
Out: $\tilde{\boldsymbol{d}}'^{(p)} \in R_{Q_{l-1},N,p}, \tilde{\boldsymbol{d}}'^{(m)} \in R_{Q_{l-1},N,m}$
 1: $\boldsymbol{t}^{(p)} \leftarrow \mathtt{INTT_p}(\tilde{\boldsymbol{d}}^{(p)}[l-1])$, $\boldsymbol{t}^{(m)} \leftarrow \mathtt{INTT_m}(\tilde{\boldsymbol{d}}^{(m)}[l-1])$
 2: $\boldsymbol{t} \leftarrow \mathtt{Join}(\boldsymbol{t}^{(p)}, \boldsymbol{t}^{(m)})$
 3: for $i = 0$ to $l - 2$ do
 4:     $\boldsymbol{r}^{(p)}, \boldsymbol{r}^{(m)} \leftarrow \mathtt{Split}([\boldsymbol{t}]_{q_i})$
 5:     $\tilde{\boldsymbol{r}}^{(p)} \leftarrow \mathtt{NTT_p}(\boldsymbol{r}^{(p)})$, $\tilde{\boldsymbol{r}}^{(m)} \leftarrow \mathtt{NTT_m}(\boldsymbol{r}^{(m)})$    ▷ in $\mathbb{Z}_{q_i}$
 6:     $\tilde{\boldsymbol{d}}'^{(p)}[i] \leftarrow [q_l^{-1} \odot (\tilde{\boldsymbol{d}}^{(p)}[i] - \tilde{\boldsymbol{r}}^{(p)})]_{q_i}$, $\tilde{\boldsymbol{d}}'^{(m)}[i] \leftarrow [q_l^{-1} \odot (\tilde{\boldsymbol{d}}^{(m)}[i] - \tilde{\boldsymbol{r}}^{(m)})]_{q_i}$
 7: end for
```

polynomial rings $\mathbb{Z}_Q[x]/x^N - \zeta_{4N}^N$ and $\mathbb{Z}_Q[x]/x^N + \zeta_{4N}^N$. The NTT and INTT operations over these different polynomial rings are performed using different twiddle constants. We use $\mathtt{NTT_p}/\mathtt{INTT_p}$ and $\mathtt{NTT_m}/\mathtt{INTT_m}$ to represent the NTT/INTT operations over the two rings $\mathbb{Z}_Q[x]/x^N - \zeta_{4N}^N$ and $\mathbb{Z}_Q[x]/x^N + \zeta_{4N}^N$, respectively.

4 Architecture of Medha

4.1 Parameter Selection

To implement RNS-HEAAN with the proposed flexible design method, we use 60 and 54-bit prime moduli as RNS bases similar to SEAL [SEA20] and HEAX [RLPD20] with each prime satisfying $q_i \equiv 1 \pmod{4N}$. For proof of the concept, Medha is implemented

Table 1: Parameter Sets

Param. Set	$(\log_2(pQ), N)$	$L+1$	**Mul. Depth**	**Sec. Level**
Set-1	$(438, 2^{14})$	8	7	>128-bit
Set-2	$(546, 2^{15})$	10	9	>128-bit

to support two large parameter sets as shown in Table 1. It supports two polynomial degrees, $N = 2^{14}$ and $2N = 2^{15}$, with up to 546-bit coefficient modulus size using 10 prime moduli. Next, we present the low-level arithmetic units of the Medha.

4.2 Low-level Arithmetic Units

In this section, we present the low-level arithmetic units of Medha starting from the lowest level of the implementation hierarchy (Fig. 1) containing 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 [SRTJ+19, RLPD20], 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. See Appendix A for details.

4.2.1 Parallel NTT Architecture

The next level of Fig. 1, performs the arithmetic of large-degree polynomials. Polynomial multiplication is the most expensive operation and it can be implemented efficiently using NTT-based polynomial multiplication. The NTT unit has been designed and optimized for $N = 2^{14}$.

Our NTT-based multiplier uses an iterative NTT approach. It utilizes the decimation-in-time (DiT) approach for the forward NTT and decimation-in-frequency (DiF) approach for the inverse NTT (INTT) [Sco17]. This particular combination eliminates the need for any coefficient permutations before the NTTs or after the INTT. The DiT NTT Algorithm is presented in Algorithm 10 in Appendix B.

For fast polynomial multiplications, we implement a multi-core parallel NTT unit borrowing the best practices from [RJV+18, SRTJ+19, RLPD20] including the routing and BRAM access optimizations from [RJV+18]. In addition, we apply an 'address delaying' 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 compute 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 is optimized for polynomials of $N = 2^{14}$ coefficients and it uses 16 compute cores. It computes one 2^{14}-point NTT in around 7,168 cycles.

4.2.2 Unified Compute Core for DiT NTT and DiF INTT

We design a single NTT unit that employs unified compute cores for the DiT NTT and DiF INTT. 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 $(u'', t'') \leftarrow (\frac{u+t}{2}, \frac{(u-t)\cdot\omega}{2})$. These operations are referred to as the butterfly operations. Our unified compute core uses one modular multiplier, one modular adder, two modular subtractors and a few two-to-one multiplexers. The modular multiplier is heavily pipelined

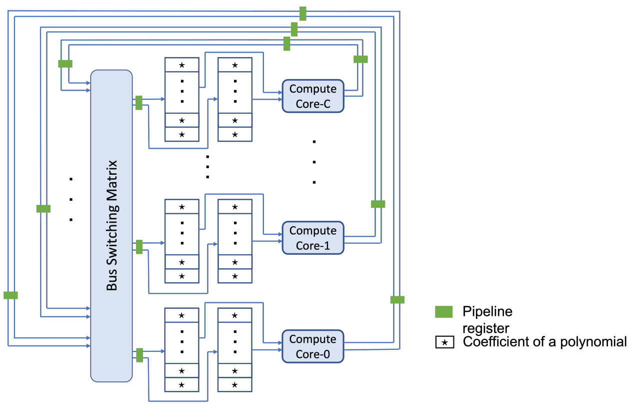

Figure 2: Organization of memory and compute cores for efficient implementation of parallel NTT

(around 20 stages) to achieve high clock frequency. From now on, we refer to compute core as the butterfly core.

Address delaying optimization: We propose an 'address delaying' optimization that causes a significant reduction in the register consumption of the pipelined butterfly cores. 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 $(u + t \cdot \omega, u - t \cdot \omega)$, u and $\omega \cdot t$ must reach the inputs of the adder and subtractor synchronously in the same cycle. In [RJV+18], 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 by simply delaying the read of coefficient u from the memory. This technique saves around 1,200 registers per butterfly unit and around 192,00 registers for overall architecture. We keep u and t in separate memory elements as shown 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.

4.2.3 Twiddle Factors during NTT

Optimized software implementations (e.g., SEAL [SEA20]) and also the hardware implementations [SRTJ+19, RLPD20] save cycle count of NTT by keeping all twiddle constants in large tables. As previous hardware implementations [SRTJ+19, RLPD20] indicated that HE 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. Each NTT butterfly unit is coupled with a twiddle factor generation unit which mainly consists of a modular multiplier and two BRAM36K memory units for storing a few initial twiddle factor constants. The multiplier in the twiddle factor generation unit is not an additional cost as it is reused to parallelize coefficient-wise multiplication and modular reduction operations.

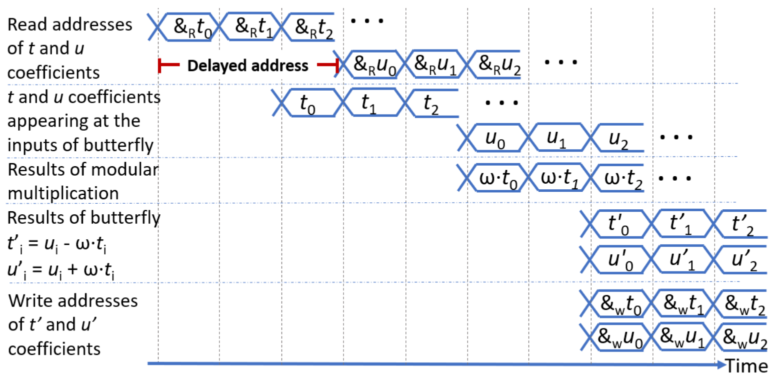
Read addresses of t and u coefficients
$\&_R t_0$ $\&_R t_1$ $\&_R t_2$ ...
Delayed address
$\&_R u_0$ $\&_R u_1$ $\&_R u_2$ ...
t and u coefficients appearing at the inputs of butterfly
t_0 t_1 t_2 ...
u_0 u_1 u_2 ...
Results of modular multiplication
$\omega \cdot t_0$ $\omega \cdot t_1$ $\omega \cdot t_2$ ...
Results of butterfly
$t'_i = u_i - \omega \cdot t_i$
$u'_i = u_i + \omega \cdot t_i$
t'_0 t'_1 t'_2
u'_0 u'_1 u'_2
Write addresses of t' and u' coefficients
$\&_w t_0$ $\&_w t_1$ $\&_w t_2$
$\&_w u_0$ $\&_w u_1$ $\&_w u_2$
Time

Figure 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.

4.2.4 Building Blocks for Supporting Parameter Flexibility

The proposed flexible design method requires very few specific low-level building blocks, which can be implemented easily.

Split and Join operations: Although they can be performed by a software system (master mode), we implement them in the hardware to avoid unnecessary HW-SW data transfers. A naive approach is utilizing separate computation units for these operations. Instead, for implementing the split and join operations efficiently, we leverage the resemblance of split and join operations to DiT and DiF butterfly structures respectively, and implement the split and join operations without any additional hardware cost using the butterfly cores. Since the NTT unit has 16 butterfly cores, each of `Split` and `Join` takes ≈1,024 clock cycles.

On-the-fly twiddle factor generation: As explained in Sec. 3.2, the proposed architecture should support N-point NTT and INTT operations for three different polynomial rings: $\mathbb{Z}_Q[x]/x^{2^{14}}+1$ for the parameter with the polynomial degree $N = 2^{14}$; $\mathbb{Z}_Q[x]/x^{2^{14}} + \zeta_{4N}^{2N}$ and $\mathbb{Z}_Q[x]/x^{2^{14}} - \zeta_{4N}^{2N}$ for the parameter with the polynomial degree $2N = 2^{15}$. The ring-specific NTTs use different twiddle constants. As described earlier, our twiddle factor generation unit stores only a few initial constants and starting from them computes the remaining constants on-the-fly. As there are three rings, three sets of initial twiddle constants are required to be stored in the memory. That will increase the on-chip memory requirement. Instead, we store only the initial twiddle constants for the polynomial rings $\mathbb{Z}_Q[x]/x^{2^{14}} - \zeta_{4N}^{2N}$ and $\mathbb{Z}_Q[x]/x^{2^{14}} + \zeta_{4N}^{2N}$. When an NTT/INTT for the ring $\mathbb{Z}_Q[x]/x^{2^{14}}+1$ is to be computed, the required initial twiddle constants are generated by multiplying those for the ring $\mathbb{Z}_Q[x]/x^{2^{14}} - \zeta_{4N}^{2N}$ with ζ_{4N}^{i} where i changes with the NTT stages.

4.3 Architecture of the Homomorphic Processor

We take the optimized polynomial arithmetic units from the previous section and organize them in the architecture of Medha. Medha is an instruction-set architecture (ISA) and therefore programmable to run the homomorphic subroutines, namely addition/subtraction, multiplication, key-switching, relinearization and rescaling flexibly by reusing the same compute units. Previous works [RJV+18, SRTJ+19] presented ISAs for accelerating homomorphic encryption, but their performance advantages remained limited,

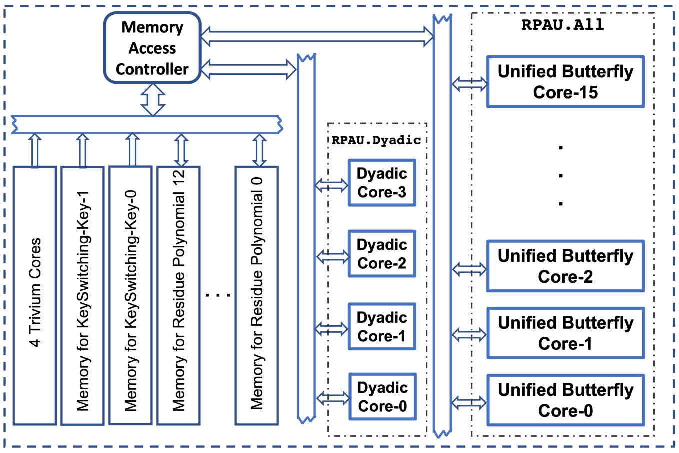

Figure 4: Architecture of the Residue Polynomial Arithmetic Unit (RPAU).

e.g., only one order speedup [SRTJ$^+$19] compared to SW implementations. On the contrary, HEAX [RLPD20] organized its polynomial arithmetic elements in a subroutine-specific manner to implement a key-switching unit that gives over two orders higher throughput compared to SW. The one order speedup of HEAX over [SRTJ$^+$19] demonstrated that block-pipelined and subroutine-specific HE accelerators are much superior to programmable accelerators that reuse compute elements.

We organize compute and memory elements pragmatically and realize a *flexible* and *programmable* accelerator that achieves up to 78× performance improvement compared to Microsoft SEAL [SEA20]. Our work shows that HE accelerators do not have to sacrifice programmability to achieve high speed. The following subsections describe how we organize compute and memory elements.

4.3.1 Design of Residue Polynomial Arithmetic Unit (RPAU)

In any RNS-based HE, 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 [RJV$^+$15], 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. 2) 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.

As shown in Algorithm 2 and Algorithm 5, some of the steps in `HEAAN.Mult` and `HEAAN.Relin` can be performed in parallel. For example, calculation of $\tilde{\boldsymbol{d}}_0$, $\tilde{\boldsymbol{d}}_1$ and $\tilde{\boldsymbol{d}}_2$, `NTT`($\tilde{\boldsymbol{d}}_2$) can performed using `RPAU.Dyadic` and `RPAU.All`, respectively, in parallel. Inside

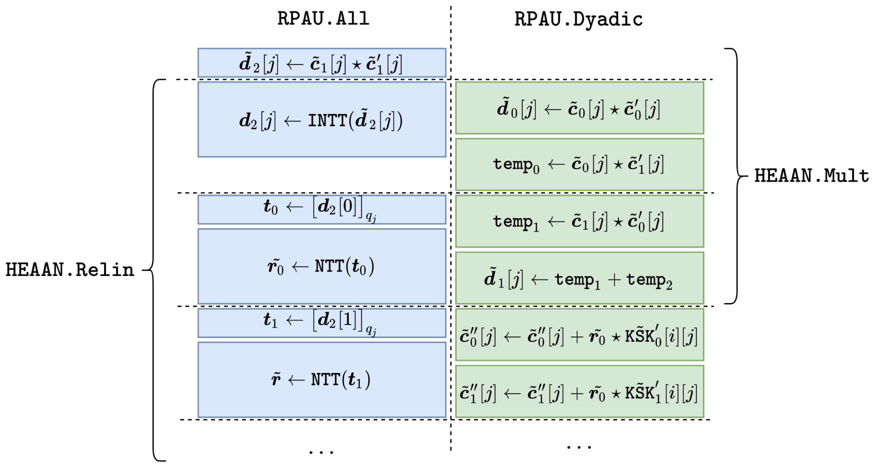
RPAU.All
RPAU.Dyadic
$\tilde{d}_2[j] \leftarrow \tilde{c}_1[j] \star \tilde{c}'_1[j]$
$d_2[j] \leftarrow \mathtt{INTT}(\tilde{d}_2[j])$
$\tilde{d}_0[j] \leftarrow \tilde{c}_0[j] \star \tilde{c}'_0[j]$
$\mathtt{temp}_0 \leftarrow \tilde{c}_0[j] \star \tilde{c}'_1[j]$
$t_0 \leftarrow [d_2[0]]_{q_j}$
$\tilde{r_0} \leftarrow \mathtt{NTT}(t_0)$
$\mathtt{temp}_1 \leftarrow \tilde{c}_1[j] \star \tilde{c}'_0[j]$
$\tilde{d}_1[j] \leftarrow \mathtt{temp}_1 + \mathtt{temp}_2$
$t_1 \leftarrow [d_2[1]]_{q_j}$
$\tilde{r} \leftarrow \mathtt{NTT}(t_1)$
$\tilde{c}''_0[j] \leftarrow \tilde{c}''_0[j] + \tilde{r_0} \star \mathtt{K\tilde{S}K}'_0[i][j]$
$\tilde{c}''_1[j] \leftarrow \tilde{c}''_1[j] + \tilde{r_0} \star \mathtt{K\tilde{S}K}'_1[i][j]$
HEAAN.Mult
HEAAN.Relin
...
...

Figure 5: Parallel processing of `HEAAN.Mult` and `HEAAN.Relin` using two threads inside j-th RPAU.

the loop of the key-switching, the steps are sequential due to data dependencies. As the loop iterates several times, we can unroll 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 shown in steps 4-10 of Algorithm 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 7,168 cycles) and effectively reduce the latency of steps 7-8 in Algorithm 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. 5 gives a timing diagram and shows how we can speedup the computation of `HEAAN.Mult` and `HEAAN.Relin` using the two parallel compute groups.

4.3.2 Organization of On-chip Memory inside RPAU

Peak memory requirement: After optimizing the steps of homomorphic multiplication followed by relinearization, we observe that the peak memory requirement per RPAU is equal to storing seven residue polynomials (ciphertext-dependent data), and additional $2L$ residue polynomials of the key-switching-key. E.g., for Set-1 we have $L = 7$ and therefore we need to store 21 residue polynomials on-chip for each RPAU to eliminate off-chip memory access completely during key-switching. In a similar way, the peak memory requirement for Set-2 with $2N = 2^{15}$ and $L = 9$ becomes equivalent to storing 49 residue polynomials of degree $N = 2^{14}$ per RPAU. To handle such large data, an option will be to use an off-chip memory. That will involve the cost of slow data exchanges. Note that the proposed parameter-flexible design methodology makes such a data-partitioning easy to implement.

We take a step further and aim at keeping all the data in the on-chip memory for both Set-1 and 2. Neither BRAMs nor URAMs alone in U250 could store such a big data, but their combination with a few tricks can.

On the fly key-switching key generation: The large size of the key-switching key increases the peak memory requirement significantly. We notice that the $\texttt{KSK}_0$ component of the key-switching key $\texttt{KSK}$ is generated by expanding a pseudorandom public seed. The other component $\texttt{KSK}_1$ is computed using the secret key. Therefore, the cloud can regenerate $\texttt{KSK}_0$ itself from the seed. We use that to avoid storing half of the key-switching key by generating the pseudorandom polynomials of $\texttt{KSK}_0$ on-the-fly using a PRNG in the hardware. Note that similar techniques are used in lattice-based post-quantum algorithms, example [DKRV19], to generate their public polynomials.

In our implementation, we use the 64-bit Trivium core [Bou20]. The trivium core takes a 64-bit seed as input, performs 18 round initialization in 18 clock cycles, and thereafter generates 64-bit random data per cycle without any stall. We use four Trivium cores per RPAU as there are four dyadic cores too per RPAU. These dyadic cores consume the Trivium$\times$4 generated pseudorandom words.

Overall, on the fly generation of $\texttt{KSK}_0$ reduces the storage space for the key-switching key by half. For $2N = 2^{15}$ and $L = 9$ now we need to store only 31 2^{14}-degree residue polynomials in the on-chip memory per RPAU.

Memory unit: We keep all the data variables for both the parameter sets Set-1 and 2 in the designed on-chip memory unit. We are the first to achieve fully on-chip computation of the key-switching operation for the two large parameters. On the target U250 FPGA, one URAM is 8 times larger than one BRAM36K. However, both types of memory have only two ports. Quantitatively, one 2^{14}-degree 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 during an NTT/INTT operation. Hence, designing the memory unit of the RPAU requires careful consideration 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. 6. The memory-bank of the i-th core of NTT keeps only the i-th fragments of all the 31 residue polynomials. In Fig. 6 the abbreviation 'RPM' stands for the residue polynomial memory. There are 13 RPMs as there are 13 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-8, 9, and 10 are composed of BRAM36K slices and are physically separated. Hence, they can be read/written in parallel. Whereas,

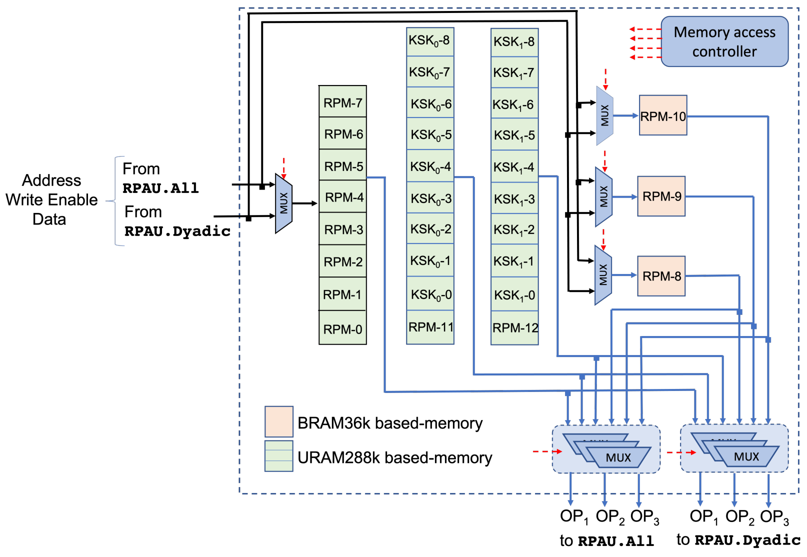

Figure 6: Organization of memory elements inside the memory bank. One memory bank is connected to a one core of the NTT unit. There are 16 such memory banks inside each RPAU. Any residue polynomial is split into 16 fragments, and one fragment is stored in one RPM-i of a memory module.

RPM-0-to-7 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. Also, URAM-based RPM-11 and 12 are reserved for ciphertext-dependent residue polynomials. Programmer decides which polynomial goes to which RPM taking care of data dependencies and access patterns of the HE subroutine. The 'Memory access controller' block is responsible for handling memory accesses of the two parallel computing groups `RPAU.All` and `RPAU.Dyadic`.

We also use URAMs to store key-switching key. We need to store $2L = 18$ 2^{14}-degree residue polynomials for key-switching key, which require 72 URAMs. Each address of a URAM can store 72-bit. For a 54-bit coefficient modulus, one 2^{14}-degree residue polynomial can be stored in $(54 \cdot 4)/72 = 3$ URAMs instead of 4 by utilizing the left-over bits in each URAM address. Therefore, for RPAU units with 54-bit coefficient modulus, we only use 56 URAMs instead of 72 to store key-switching key. Similarly, we use 63 URAMs instead of 72 for 60-bit coefficient modulus.

4.3.3 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 HE 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 HE 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 [Xilb], our Alveo U250 platform uses SSI technology and consists of four 'semi-separated' SLR regions. Two neighbouring SLRs are

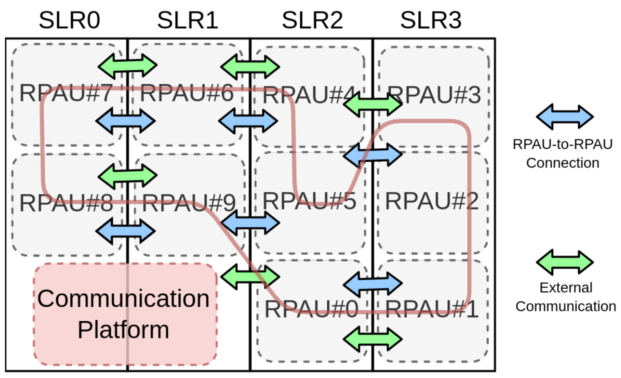

Figure 7: The proposed 'ring' structured floorplan to minimize routing cost for the implementation with 10 RPAU units.

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-to-SLR connection constraint into account to make implementation feasible. Additionally, the communication unit between of the FPGA resides in SLR0 and 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. 7, we show placement of 10 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. 7.

4.3.4 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 mod-down and rescaling operations, 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.

4.3.5 Scheduling of Homomorphic Operations

The proposed Medha is an instruction-set architecture. Therefore, efficient translation of homomorphic operations into Medha instructions is crucial for achieving optimal performance results. In this section, we describe the scheduling of homomorphic operations for both parameter sets.

Scheduling for $N = 2^{14}$: `HEAAN.Add` operation is straightforward and it can easily be implemented using two coefficient-wise addition instructions. The `HEAAN.Mult` operation can be performed using coefficient-wise multiplication and addition instructions. As shown in Algorithm 5, relinearization operation requires each residue polynomial in $\tilde{\boldsymbol{d}}_2$ to be reduced by the other moduli (see step 7 of Algorithm 5). Hence, all residue polynomials in $\tilde{\boldsymbol{d}}_2$ first need to be transformed to the time-domain using one INTT instruction in the parallel RPAUs as shown in steps 1-3 of Algorithm 5. Since `HEAAN.Relin` operation needs $\tilde{\boldsymbol{d}}_2$ output of `HEAAN.Mult` first, `HEAAN.Mult` and `HEAAN.Relin` operations can be performed efficiently in parallel. As soon as $\tilde{\boldsymbol{d}}_2$ is generated, INTT operation is performed using the main computation core while remaining outputs of `HEAAN.Mult` can be generated using dyadic cores in parallel. Then, the i-th sends the i-th residue polynomial in $\boldsymbol{d}_2$ to the other RPAUs through the ring structure. This operation is performed using an instruction that broadcasts a residue polynomial in an RPAU to other RPAUs. The received $\boldsymbol{d}_2[j]$ 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 Algorithm 5. Note that, the coefficient-wise multiplications and accumulations in the key-switching loop (steps 7-8 in Algorithm 5) 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 $l-1$ NTTs. `HEAAN.ModDown` and `HEAAN.Rescale` operations are implemented sequentially using INTT, NTT and broadcasting instructions of the main core as they are not parallelizable in instruction-level.

Scheduling for $2N = 2^{15}$: The scheduling of homomorphic operations for the parameter size $2N = 2^{15}$ is very similar to the scheduling for the parameter size $N = 2^{14}$ except for the routines involving `Split` and `Join` functions. The `HEAAN.Add` and `HEAAN.Mult` operations can be performed efficiently using coefficient-wise addition and multiplication instructions on the split ciphertexts. We first store all split ciphertexts $\texttt{ct}^{(p)}$ and $\texttt{ct}^{(b)}$ into the on-chip memory, then perform homomorphic operations $\texttt{ct}^{(p)}$. After that, we move $\texttt{ct}^{(m)}$ into the proper memory locations and repeat the same homomorphic operations.

As shown in Algorithm 8, the relinearization operation requires split/join and therefore $\texttt{ct}^{(p)}$ and $\texttt{ct}^{(m)}$ cannot be processed completely independently. As a consequence, `HEAAN.Relin` requires some extra on-chip data movement for split/join operations during the computations. Due to split/join and extra on-chip data movements, the homomorphic operations for the parameter size $2N = 2^{15}$ costs more than $2\times$ clock cycles than the homomorphic operations for the parameter size $N = 2^{14}$.

4.3.6 Hardware-Software Interfacing of the Overall System

We implemented a proof-of-concept software stack (Fig. 8) 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

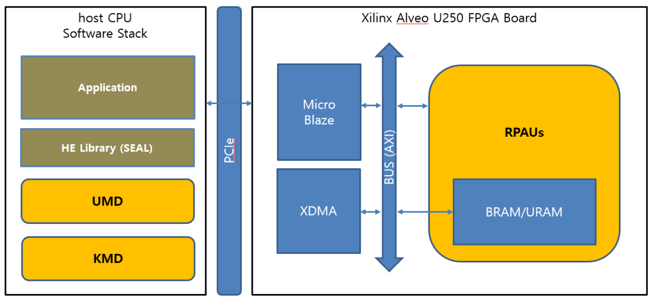

Figure 8: CPU-FPGA interface and software stack

result is read through the PCIe interface. All data communications are performed using XDMA [Xilc] 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.

5 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 implementation of Medha employs 10 RPAU units and works for the Set-1 and Set-2 from Table 1.

5.1 Timing Results

The proposed implementation runs at 200MHz. In Table 2, we present the cycle count, latency (in μ sec) and throughput results for each low-level instruction and high-level operations for the Set-1 and Set-2. The cycle counts were collected using a hardware-based counter. 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 2. 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.

5.2 Resource Utilization Results

Table 3 shows the resource requirements for the complete processor architecture, one RPAU (with 54-bit and 60-bit modulus), one butterfly core, and one twiddle factor generation core. The area figures for the complete processor also include the 'Platform' unit that is responsible for the communication between the FPGA and the host CPU. The overall design consumes 55.4% of LUTs, 21.2% of registers, 58.6% of BRAMs, 72.7% of URAMs and 29.3% of DSPs available on the Alveo U250 Card. Since we target performing homomorphic operations using only on-chip memory, the proposed design has very high BRAM and URAM utilization. Although the resulting URAM utilization allows instantiating more RPAU units, there are several reasons preventing us from adding more RPAU units. Firstly, the 'Platform' unit consumes more resources than one RPAU unit and it resides across two SLR regions, which limits the placement of new RPAUs. Secondly, increasing the number of RPAU units (hence the number of RNS bases) will increase the size of key-switching key. Therefore, each RPAU unit will employ more URAMs to store keys. Since each SLR region has only 320 URAMs in Alveo U250 Card, placing 3 RPAUs into a single SLR region will be much harder. Also, high routing cost of SLR-to-SLR communication prevents placing

Table 2: Performance of each Instruction/Operation

Set	Instruction/Operation	Clock Cycles*	Lat. (μ sec)	Throug. (per sec)
Set-1 & Set-2	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
	Split/Join	$\approx$ 1,024	5.12	195,312
Set-1	Hom. Add	1,152	5.76	173,611
	Hom. Mult. + Relin.	99,448	497.24	2,011
	Rescale	34,430	172.15	5,808
Set-2	Hom. Add	2,865	14.33	69,783
	Hom. Mult. + Relin.	274,885	1,374.42	727
	Rescale	75,464	377.32	2,650

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

Table 3: Resource Utilization of Arithmetic Modules on Alveo U250 Card

Modules	LUTs	REGs	BRAMs	URAMs	DSPs
Processor	958,041	733,598	1,576.5	931	3,607
⌊Platform	128,334	132,146	296.5	1	7
⌊Prog. Controller	851	276	–	–	–
⌊Core	827,829	599,182	1280	930	3,600
RPAU Unit (54-bit)	79,382	59,617	128	92	360
⌊Memory Core	23,082	504	96	92	–
⌊$\texttt{KSK}_0$ Core	1,936	1,594	–	–	–
⌊Dyadic Core	5,674	3,567	–	–	40
⌊Main Core	35,370	41,240	32	–	320
RPAU Unit (60-bit)	83,491	60,018	128	102	360
⌊Memory Core	23,706	164	96	102	–
⌊$\texttt{KSK}_0$ Core	2,030	1,618	–	–	–
⌊Dyadic Core	7,212	3,699	–	–	40
⌊Main Core	42,324	2,636	32	–	320
Butterfly Unit	1,358	1,625	–	–	10
⌊Modular Mult.	533	782	–	–	10
TF Gen. Unit	755	962	2	–	10

one RPAU across multiple SLR regions. Finally, adding a new RPAU complicates the routing and decreases the maximum achievable clock frequency of the overall design. Thus, we set the number of RPAU units as 10.

Alveo U250 FPGA card can have a 215W maximum power consumption level [Xila] with a typical consumption level of around 100W. For Medha, Vivado reported on-chip power consumption of 62.12W.

5.3 Comparison with Related Works

Comparison with SEAL: There are various highly-optimized software implementations of the HEAAN scheme based on homomorphic encryption libraries such as Microsoft

Table 4: Latency Comparison with the SEAL [SEA20] and HEAX [RLPD20]

Set	Work	Add	Mult. + Relin.	Rescale
Set-1	**Medha**	5.76 μs	497.24 μs	172.15 μs
	SEAL [SEA20]	418 μs (**72.56**×)	33,844 μs (**68.06**×)	6,429 μs (**37.34**×)
	SEAL [RLPD20]	– –	66,666 μs (**134.07**×)	– –
	HEAX [RLPD20]	– –	1,182.27 μs (**2.37** ×)	– –
Set-2	**Medha**	14.33 μs	1,374.42 μs	377.32 μs
	SEAL [SEA20]	921 μs (**65.03**×)	108,401 μs (**78.87**×)	16,438 μs (**46.56**×)

SEAL [SEA20] and Palisade [PRR17]. 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` [SEA20]. To present a fair comparison, we modified the SEAL accordingly to work with the parameter sets defined in Table 1. The latency of high-level homomorphic operations in SEAL [SEA20] and its comparison to Medha for Set-1 and Set-2 are presented in Table 4. 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 and Set-2 showed 68.06× and 78.87× performance improvements, respectively, for the homomorphic multiplication with relinearization operation compared to the SEAL-based implementation. The effectiveness of our approach increases with the larger parameter sets. In Table 4, 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 [RLPD20], for the Set-1 parameter. Compared to this result, Medha shows 134.07× speed-up for homomorphic multiplication and relinearization operations.

Comparison with HEAX: The fairest comparison is with the HEAX processor [RLPD20]. It is the only prior art for actual FPGA-based implementation of RNS-HEAAN and with the same parameter set (Set-1). 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-1 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 2,011 homomorphic multiplications including key-switching operations per second while running at a lower clock frequency of 200MHz for the Set-1 parameter.

Latency-wise, Medha is more than 2× faster than HEAX as shown in Table 4 for the Set-1 parameter. As the latency figures of HEAX are not specified in [RLPD20], we estimate them based on the computation flow diagram from Table 5 and Figure 6 of [RLPD20] as follows. There are six stages of block-pipeline processing during a key-switching (the last row or Set-C of Table-5 in [RLPD20]) 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 [RLPD20]), computing a full key-switching will take at least 358,400 cycles. In comparison, our Medha has a latency of 99,448 cycles only for computing one homomorphic multiplication plus a key-switching for the Set-1 parameter.

From an architect's perspective, Medha has five 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-1, (*iii*) HEAX uses a fixed-pipelined architecture tailored for HEAAN scheme while Medha's instruction-based architecture allows flexibility, (*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 [RLPD20]), and (*v*) HEAX provides support for polynomial degree 2^{14} while Medha supports homomorphic operations for multipler parameters with $N = 2^{14}$ and $2N = 2^{15}$. 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 [RJV+18, SRTJ+19] would have indicated that specific and block-pipelined processors are the way to accelerate HE in HW.

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 for the Set-1 parameter 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. For example, consider computing the e^5 from an encryption of e. 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).

Comparison with F1: [FSK+21] proposed an instruction-based wide-vector processor architecture 'F1' and presented RTL simulation-based performance estimates using cycle accurate C/C++ models. 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-1 parameter. However, latency figures for homomorphic operations are not presented in F1. Although F1 supports multiple polynomial degrees, it provides support only up to polynomial degree 2^{14} while our Medha supports the polynomial degrees 2^{14} and 2^{15}.

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 [GFS+12] 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 for $N = 2^{14}$. 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.

Note that there is no real ASIC hardware of F1. 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 [RJV+18, SRTJ+19, TRV20] present the FPGA implementations for the high-level operations of the BFV scheme. In [RJV+18], 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 [SRTJ+19] and [TRV20] 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-based 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 [RTJ+20, TRG+20]. As these works target 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 [JKA+21, BHM+20]. For the parameter set $N = 2^{14}$ and $\log_2 Q = 360$, [BHM+20] performs homomorphic addition and homomorphic multiplication with relinearization in 0.04 ms and 0.74 ms, respectively. For a similar parameter set (Set-1), our architecture shows 6.9× and 1.5× 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 with relinearization, respectively. Also, their multi-GPU platform NVIDIA DGX-1 has a reported maximum power consumption level of 3,500W which is 16× higher compared to the Alveo U250 board (at peak power). The NVIDIA DGX-1 platform costs $49,000 which is 6.3× higher than the Alveo U250 board. Therefore, Medha running on an Alveo U250 board would be a more power-efficient and cheaper accelerator solution for homomorphic applications.

The work in [JKA+21] 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) [DS15, BVMA18, BPA+18, BCL+20] or partial operations such as NTT [ZIQ+21, ÖEM+22].

6 Discussions

Implementation challenges: The biggest challenge we faced during the implementation was finding an efficient placement of the proposed design on an actual FPGA board. In a large design, although individual building blocks achieve high clock frequency, the overall design still can end up with a low clock frequency if the placement of building blocks is not performed carefully. It took several months and many iterations for us to find an efficient placement of RPAU units and a way to move data across a large FPGA, which yields high clock frequency. The first step was detecting problematic timing paths which happened to be the data signals for external communication routing from SLR1 to other SLR regions and placing extra pipeline buffers in these paths for providing flexibility in routing and improving the clock frequency. However, these additional buffers showed little or no improvements for the clock frequency due to the inefficient placement of RPAU units. Finally, the 'ring-like' interconnection detailed in Sec. 4.3.3, where each RPAU is constrained to a certain SLR region, made it possible to achieve a high clock frequency. These real engineering challenges do not appear in a high-level or RTL simulation of architectures.

Performance/cost estimate for larger parameters: In this paper, we implemented Set-1 and 2 and both implementations do not require off-chip communications during a key-switching. For larger parameter sets, it will no longer be possible to avoid off-chip data transfers. In such implementations, the proposed flexible design method will be particularly useful as it enables the re-use of smaller arithmetic units and memory. Now, we present estimates for larger parameter sets assuming that the resources of the target FPGA scale appropriately. For $N = 2^{16}$ and $L = 9$, the peak memory requirement for the ciphertext-dependent data and key-switching key will be 25 and $4L = 36$ residue polynomials, respectively. Following the same practices from Sec. 4.3.2, each RPAU will require 96 BRAMs and 208 URAMs. Then, the overall design will consume more than $2\times$ more URAMs.

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 [Xilb] to combine multiple Super Logic Regions (SLR). Therefore, the proposed ring of RPAUs (Sec. 4.3.3) will be essential to implementing multi-RPAU HE 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 and 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.

Benchmarking of homomorphic applications: Transforming plaintext applications into *optimized* homomorphic applications is generally not straightforward [DSC+19]. In this work, we focused on the processor architecture design and considered HE application development and benchmarking as future works. Similar to HEAX [RLPD20] and F1 [FSK+21], we provide estimates of application benchmarking.

Our parameter sets are sufficient for evaluating logarithmic and exponential functions which are used for approximate computations [KR20]. PrivFT implementation [BHM+20] 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 $\log_2 pQ = 360$ (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× speedup over the GPU implementation of PrivFT [BHM+20].

In [BCIV17], 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× 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.

7 Conclusion

Despite being theoretically sound, HE suffers from performance issues due to its massive computational costs. Hence, hardware accelerators are crucial for making HE fast and practical. Also, providing support for multiple scheme parameters is another important requirement for hardware accelerator as different applications demand different scheme parameters. In this paper, we first propose a flexible design methodology for the polynomial degree. It enables supporting multiple polynomial degrees using a fixed hardware architecture. Then, we propose Medha, a flexible and programmable accelerator architecture that has been designed pragmatically to overcome the speed and flexibility limitations of previous HE 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 large HE parameters. Medha also utilizes the proposed flexible design methodology for the polynomial degree and it supports polynomial degrees 2^{14} and 2^{15} with coefficient modulus up to 546 bits. Medha presents the first FPGA-based hardware accelerator for the RNS-HEAAN scheme with the polynomial degree 2^{15}.

Medha was implemented in the Xilinx Alveo U250 card and accurate performance results were obtained for two large parameter sets. Compared to the highly-optimized SEAL [SEA20] library, our Medha achieved up to 68× and 78× speedup on an Intel i5 CPU for the parameters $(\log Q, N) = (438, 2^{14})$ and $(546, 2^{15})$, respectively. Medha achieved almost 2.37× latency improvement compared to the block-pipelined and specific hardware accelerator HEAX [RLPD20] for the parameter $(\log Q, N) = (438, 2^{14})$. The speed improvement shows that hardware accelerators for HE can attain high performance without losing programmability or flexibility.

A Modular Arithmetic Architectures

For implementing modular arithmetic, 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 use the DSP units available in FPGAs so that the highest throughput can be achieved. We used several layers of pipeline registers to meet high clock-frequency constraints. One integer multiplier (60-bit or 54-bit) consumes 10 DSP slices. The most commonly used methods for performing modular reduction are based on the Barrett or Montgomery methods. The SEAL library [SEA20] and the hardware architecture of the HEAX processor [RLPD20] use the Barrett reduction technique for reducing the results of integer multiplications. Both Barrett and Montgomery methods are based on multiplications. Another method for implementing the modular reduction operation is using a table-based modular reduction approach as proposed in [RJV+18]. 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 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 [ZYC+20]. In this work, we use reduction-friendly pseudo-Mersenne primes and employ the fast add-shift-based modular reduction method to save both DSPs and LUTs [ZYC+20]. For example, the first modulus in the RNS-basis (q_0) is a 60-bit sparse prime $2^{59}+2^{25}+2^{22}-2^{20}+1$ and the result of a multiplication (120 bit) is reduced using the relation $2^{59} \equiv -2^{25} - 2^{22} + 2^{20} - 1 \pmod{q_0}$ recursively. This approach saves up to 45% LUT units compared to the table-based modular reduction method.

B NTT Algorithm

Algorithm 10 Decimation-in-time (DIT) Forward NTT

In: $\boldsymbol{a} \in R_{Q,N}$ and $\psi \in \mathbb{Z}_q$ ($2N$-th root of unity)
Out: $\tilde{\boldsymbol{a}} = \texttt{NTT}(\boldsymbol{a}) \in R_{Q,N}$

```
 1: $\tilde{\boldsymbol{a}} \leftarrow \texttt{BitReverse}(\boldsymbol{a})$          ▷ Permutation of coefficients
 2: for $m = 2$ to $N$ by $m = 2m$ do
 3:     $w \leftarrow 1$, $\psi_m \leftarrow \psi^{N/m}$
 4:     for $j = 0$ to $m/2 - 1$ do                          ▷ Butterfly loop
 5:         for $k = 0$ to $N - 1$ by $m$ do
 6:             $u \leftarrow \tilde{\boldsymbol{a}}[k+j], t \leftarrow \tilde{\boldsymbol{a}}[k+j+m/2]$
 7:             $\tilde{\boldsymbol{a}}[k+j] \leftarrow u + w \cdot t$
 8:             $\tilde{\boldsymbol{a}}[k+j+m/2] \leftarrow u - w \cdot t$
 9:         end for
10:         $w \leftarrow w \cdot \psi_m$
11:     end for
12: end for
```

References

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[BCIV17] Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling. In Marc Joye and Abderrahmane Nitaj, editors, *Progress in Cryptology - AFRICACRYPT 2017 - 9th International Conference on Cryptology in Africa, Dakar, Senegal, May 24-26, 2017, Proceedings*, volume 10239 of *Lecture Notes in Computer Science*, pages 184–201, 2017.

[BCL+20] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Jun Jie Sim, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. Towards the alexnet moment for homomorphic encryption: Hcnn, thefirst homomorphic cnn on encrypted data with gpus, 2020.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electron. Colloquium Comput. Complex.*, page 111, 2011.

[BHM+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.

[Bou20] Charles Bouillaguet. Trivium. https://github.com/cbouilla/trivium, December 2020.

[BPA+18] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt 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.

[BVMA18] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):70–95, May 2018.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368. Springer, 2018.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *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*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Proposal to NIST PQC Standardization, Round2, 2019. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions.

[DS15] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.

[DSC+19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 142–156. ACM, 2019.

[FSK+21] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version), 2021.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[Gen09] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.

[GFS+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin 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*, CHES'12, page 512–529, Berlin, Heidelberg, 2012. Springer-Verlag.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):114–148, Aug. 2021.

[KR20] Shabnam Khanna and Ciara Rafferty. Accelerating homomorphic encryption using approximate computing techniques. In Pierangela Samarati, Sabrina De Capitani di Vimercati, Mohammad S. Obaidat, and Jalel Ben-Othman, editors, *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications, ICETE 2020 - Volume 2: SECRYPT, Lieusaint, Paris, France, July 8-10, 2020*, pages 380–387. ScitePress, 2020.

[ÖEM+22] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Efficient number theoretic transform implementation on gpu for homomorphic encryption. *The Journal of Supercomputing*, 78(2):2840–2872, 2022.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015.

[PRR17] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. Palisade lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep*, 15, 2017.

[RAD78] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[RJV$^+$15] S. Sinha 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.

[RJV$^+$18] S. Sinha 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.

[RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.

[RNH19] Dayane Reis, Michael T. Niemier, and Xiaobo Sharon Hu. A computing-in-memory engine for searching on homomorphically encrypted data. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 5(2):123–131, 2019.

[RTJ$^+$20] Dayane Reis, Jonathan Takeshita, Taeho Jung, Michael Niemier, and Xiaobo Sharon Hu. Computing-in-memory for performance and energy-efficient homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(11):2300–2313, Nov 2020.

[Sco17] Michael 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*, pages 247–258. Springer, 2017.

[SEA20] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[SRTJ$^+$19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, 2019.

[SRVM$^+$14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[TRG$^+$20] Jonathan Takeshita, Dayane Reis, Ting Gong, Michael Niemier, X. Sharon Hu, and Taeho 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.

[TRV20] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. Heaws: An accelerator for homomorphic encryption on the amazon aws fpga. *IEEE Transactions on Computers*, 69(8):1185–1196, 2020.

[Xila] Xilinx. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.

[Xilb] Xilinx. Large FPGA Methodology Guide (UG872). https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf.

[Xilc] Xilinx. Xilinx DMA IP Reference Drivers. https://github.com/Xilinx/dma_ip_drivers.

[XZH21] Guozhu Xin, Yifan Zhao, and Jun Han. A multi-layer parallel hardware architecture for homomorphic computation in machine learning. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[ZIQ+21] Yujia Zhai, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. Accelerating encrypted computing on intel gpus. *arXiv preprint arXiv:2109.14704*, 2021.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):49–72, Mar. 2020.