HEAX: High-Performance Architecture for Computation on Homomorphically Encrypted Data in the Cloud

M. Sadegh Riazi*† mriazi@ucsd.edu
Kim Laine† kim.laine@microsoft.com
Blake Pelton† blakep@microsoft.com
Wei Dai† wei.dai@microsoft.com
*University of California San Diego
†Microsoft Research

Abstract

With the rapid increase in cloud computing, concerns surrounding data privacy, security, and confidentiality also have been increased significantly. Not only cloud providers are susceptible to internal and external hacks, but also in some scenarios, data owners cannot outsource the computation due to privacy laws such as GDPR, HIPAA, or CCPA. Fully Homomorphic Encryption (FHE) is a groundbreaking invention in cryptography that, unlike traditional cryptosystems, enables computation on encrypted data without ever decrypting it. However, the most critical obstacle in deploying FHE at large-scale is the enormous computation overhead.

In this paper, we present HEAX, a novel hardware architecture for FHE that achieves unprecedented performance improvement. HEAX leverages multiple levels of parallelism, ranging from ciphertext-level to fine-grained modular arithmetic level. Our first contribution is a new highly-parallelizable architecture for number-theoretic transform (NTT) which can be of independent interest as NTT is frequently used in many lattice-based cryptography systems. Building on top of NTT engine, we design a novel architecture for computation on homomorphically encrypted data. We also introduce several techniques to enable an end-to-end, fully pipelined design as well as reducing on-chip memory consumption. Our implementation on reconfigurable hardware demonstrates 164-268× performance improvement for a wide range of FHE parameters.

1. INTRODUCTION

Cloud computing has, in a short time, fundamentally changed the economics of computing. It allows businesses to quickly and efficiently scale to almost arbitrary-sized workloads; small organizations no longer need to own, secure, and maintain their own servers. However, cloud computing comes with significant risks that have been analyzed in the literature over the last decade (see [26, 40, 70]). Specifically, many of these risks revolve around data security and privacy. For example, data in cloud storage might be exposed to both outsider and insider threats, and be prone to both intentional and unintentional misuse by the cloud provider. Recently, the European Union and the State of California have passed strong data privacy regulations. In this light, companies and organizations that possess highly private data are hesitant to migrate to the cloud, and cloud providers are facing increasing liability concerns.

To mitigate security and privacy concerns, cloud providers should keep customers' data encrypted at all times. Symmetric-key encryption schemes, such as Advanced Encryption Standard (AES) [23], allow private data to be stored securely in a public cloud indefinitely. However, unless the customers share their secret keys with the cloud, the cloud becomes merely a storage provider. In other cases, the customers may want to securely transmit data to the cloud using protocols such as Transport Layer Security (TLS), but give the cloud access to the unencrypted data for outsourced processing. While this model unlocks the key benefits of cloud computing, it at the same time weakens the customer's data privacy compared to a cloud that only provides an encrypted storage functionality.

In 2009, a new class of cryptosystems, called Fully Homomorphic Encryption (FHE) [34], was introduced that allows arbitrary *computation on encrypted data*. This ground-breaking invention enables clients to encrypt data and send ciphertexts to a cloud that can evaluate functions on ciphertexts. Final and intermediate results are encrypted, and only the data owner who possesses the secret key can decrypt, providing an *end-to-end* encryption for the client.

FHE provides provable security guarantees without any trust assumptions on the cloud provider, and it can be used to enable several secure and privacy-preserving cloud-based solutions. For instance, in the context of Machine Learning as a Service (MLaaS), FHE can be used to perform oblivious neural network inference [36]: clients send the encrypted version of their data, the cloud server runs ML models on the encrypted queries and returns the result to the clients. All intermediate and final results are encrypted and can only be decrypted by the clients. Perhaps, the most critical obstacle today to deploy FHE at large-scale is the enormous computation overhead compared to the plaintext counterpart in which the data is not kept confidential.

Most FHE schemes, i.e., the BGV [11], the BFV [31], and the TFHE [18] perform exact computation on encrypted data. A recently proposed FHE scheme, called CKKS [17], performs approximate computation of real numbers and supports efficient *truncation* of encrypted values. Several works [51, 47] have shown the benefits of choosing the CKKS scheme over other schemes when an approximate computation is required, e.g., Machine Learning applications. Therefore, we focus on the CKKS scheme in this paper, even though our core modules are applicable to most of FHE schemes.

In this paper, we introduce HEAX (stands for Homomorphic Encryption Acceleration), a novel high-performance architecture for computing on (homomorphically) encrypted data. We design several optimized core computation blocks for fast modular arithmetic. HEAX introduces a new architecture for high-throughput Number-Theoretic Transform (NTT). NTT is a ubiquitous operation in FHE as well as many lattice-based cryptography systems. Efficient NTT engine directly improves the performance of these cryptosystems since NTT operation is usually the computational bottleneck. Building on top of NTT module, we design modules to perform high-level operations supported by FHE schemes, thus, accelerating any secure and privacy-preserving application that is based on FHE.

Prior Art and Challenges. In FHE schemes, the *ciphertext* is a set (usually a pair) of polynomials with degree $n-1$ (vector of n integers) modulo a big integer. One of the main challenges of designing an architecture for FHE is that homomorphic operations on ciphertexts involve computationally intensive modular arithmetic on big integers (with several hundred bits). These operations have convoluted data dependency among different parts of the computation, making it challenging to design a high-throughput architecture. Moreover, the degree of the underlying polynomials is enormous (in the order of several thousand). Storing the entire intermediate results on FPGA chip is prohibitive for small encryption parameters and is infeasible for large parameters, given the fact that memory consumption grows as $\mathcal{O}(n^3)$.

Prior work that propose customized hardware for non-CKKS schemes have taken one of these approaches: (i) Designing co-processors that only accelerate certain lower-level ring operations [21, 20, 74, 14, 30, 45]. High-level operations are performed on the CPU-side, which makes the co-processors of limited practical use. (ii) Storing intermediate results on off-chip memory, which significantly degrades the performance [60] to the extent that it can be worse than naive software execution [66]. (iii) Designing a hardware for a fixed modest-sized parameter, e.g., $n = 2^{12}$ [67]. However, encryption parameters determine the security-level and the maximum number of consecutive multiplications that one can perform on ciphertext, both of which are application-dependent. One of our primary design goals in HEAX is to have an architecture that can be readily used for a wide range of encryption parameters. In addition, we propose several techniques to efficiently store and access data from on-chip memory and minimize (or eliminate for some parameter sets) off-chip memory accesses.

Client-Side and Server-Side Computation. The homomorphic property of FHE enables the cloud server to perform various operations without having access to the secret key, i.e., not decrypting it. In other words, the cloud server can perform certain transformations on ciphertexts that are equivalent to performing computation on their corresponding plaintext values. For example, adding two ciphertexts results in the encryption of "summation of two underlying plaintext values". However, multiplication is significantly more complicated. Multiplication increases the number of polynomials in the ciphertext; requiring an operation, called *relinearization*, to transform the ciphertext back to a pair of polynomials. In order to benefit from SIMD-style operations, an *encoding* step is performed by the client to embed many numbers in a single ciphertext. CKKS scheme supports *rotation* in which the numbers embedded in the ciphertext can be rotated circularly (without decrypting the ciphertext).

Note that encoding, decoding, encryption, and decryption are performed on the client-side. These operations are not computationally expensive; thus, we do not implement customized hardware for these operations. Besides, it may not be realistic to assume that all clients have access to a reconfigurable hardware. The operations that are performed by the server for *evaluating* a function on ciphertexts, i.e., multiplication, relinearization, and rotation, are computationally intensive and are the focus of our work.

Contributions. In what follows, we elaborate on our major contributions in more detail:

- We design a novel architecture for number-theoretic transform (NTT) which is a fundamental building block – and usually the computation bottleneck – for many lattice-based cryptosystems including all FHE schemes. Our design is generic and can process arbitrary-sized polynomials with an adjustable throughput. We develop several techniques to overcome the challenges due to the complex data-dependency and convoluted access patterns within NTT.
- We introduce the *first* architecture for CKKS homomorphic encryption. We leverage multi-layer parallelism design starting from ciphertext-level to fine-grained optimized modular arithmetic engines. In contrast to the prior art for other FHE schemes, our architecture can be scaled for different FPGA chips due to its modularity. Moreover, HEAX is not custom-designed for specific FHE parameter set and can be used for a broad range of parameters.
- We provide a proof-of-concept implementation on two Intel FPGAs that represent two different classes of FPGAs in terms of available resources. We implement all high-level operations supported by CKKS and evaluate our design for three sets of parameters which account for the vast majority of practical applications. Our experimental results demonstrate more than *two orders of magnitude* performance improvement compared to heavily-optimized Microsoft SEAL library running on CPU.

Paper Outline. Section 2 provides the notation in the paper as well as a brief background on a few concepts. In Section 3, a detailed description of algorithms in CKKS scheme is provided. We describe our proposed architectures in Section 4, and in Section 5, we elaborate on system-view and data flow. In Section 6, resource utilization and performance of our proof-of-concept implementation is provided. We categorize and explain prior art in Section 7.

2. PRELIMINARIES

Notation. Throughout the paper, integers and real numbers are written in normal case, e.g. q. Polynomials and vectors are written in bold, e.g. $\mathbf{a}$. Vectors of polynomials and matrices are written in upper-case bold, e.g. $\mathbf{A}$. We use subscripts to denote the indices, e.g. $\mathbf{a}_i$ is the i-th polynomial or row of $\mathbf{A}$.

We assume that n is a power-of-two integer and define a polynomial ring $R = \mathbb{Z}[X]/(X^n+1)$ whose elements have degrees at most $n-1$ since $X^n = -1 \in R$. We write $R_q = R/qR$ for the residue ring of R modulo an integer q whose elements have coefficients in $[-\lfloor (q-1)/2 \rfloor, \lfloor q/2 \rfloor] \cap \mathbb{Z}$. In the actual computation, we represent coefficients in $[0,\ q-1] \cap \mathbb{Z}$. We denote by $\mathbf{u} \cdot \mathbf{v}$ the multiplication of two polynomials where the product is reduced modulo X^n+1 in R and further reduced modulo q in R_q. We denote by $\langle \mathbf{u},\ \mathbf{v} \rangle$ the dot product of two vectors, which gives $\sum_i u_i \cdot v_i$. We denote by $\mathbf{u} \odot \mathbf{v}$ the coefficient-wise multiplication $(u_0 \cdot v_0,\ u_1 \cdot v_1,\ \ldots)$.

For a real number r, $\lfloor r \rceil$ denotes the nearest integer to r (rounding upwards in case of a tie), and $\lfloor r \rfloor$ is the largest integer smaller than or equal to r. For an integer a, $[a]_p$ denotes the reduction of a modulo an integer p to $[0,\ p-1] \cap \mathbb{Z}$. These operations are also applied on a vector of real/integer numbers.

All logarithms are in base two unless otherwise indicated. We use $\mathbf{a} \leftarrow \chi$ to denote sampling $\mathbf{a}$ according to distribution χ. For a finite set $\mathbb{S}$, $U(\mathbb{S})$ denotes the uniform distribution on $\mathbb{S}$, e.g., $\mathbf{a} \leftarrow U(R_q)$ denotes sampling $\mathbf{a}$ uniformly from elements of R_q. All appearances of p denote a word-sized prime number. q denotes a product of word-sized prime numbers.

Residue Number System (RNS). There is a well-known technique to achieve asymptotic/practical improvements in polynomial arithmetic over R_q with an RNS by choosing $q = \prod_{i=0}^{L} p_i$ where p_i's are pair-wise coprime integers, based on the ring isomorphism $R_q \mapsto \prod_{i=0}^{L} R_{p_i}$. We denote the RNS representation of an element $\mathbf{a} \in R_q$ by $\overline{\mathbf{A}} = \left(\overline{\mathbf{a}}_i = [\mathbf{a}]_{p_i}\right)_{0 \le i \le L} \in \prod_{i=0}^{L} R_{p_i}$. The inverse mapping is defined based on the formula $\mathbf{a} = \sum_{i=0}^{L} \overline{\mathbf{a}}_i \pi_i \left[\pi_i^{-1}\right]_{p_i} \pmod q$, where $\pi_i = \frac{q}{p_i}$. Multiplications or additions in R_q, denoted by $\mathbf{c} = \texttt{Func}(\mathbf{a}, \mathbf{b})$, can be performed on their RNS representation: for all $i = 0, 1, \ldots, L$, compute $\overline{\mathbf{c}}_i = \texttt{Func}(\overline{\mathbf{a}}_i, \overline{\mathbf{b}}_i)$ in R_{p_i}. There exist full-RNS variants of HE schemes, such as those in [7, 37, 16], which do not require any RNS conversions except in decryption and show significant improvement over non-RNS variants. On platforms such as GPUs and FPGAs, the advantage of full-RNS variants is more significant. We can execute $\overline{\mathbf{c}}_i = \texttt{Func}(\overline{\mathbf{a}}_i, \overline{\mathbf{b}}_i)$ for all i's in parallel. Thus, the amount of memory to store temporary data is reduced to the size of R_{p_i} elements.

Gadget Decomposition. Let $\mathbf{g} \in \mathbb{Z}^d$ be a gadget vector and q an integer. The gadget decomposition, denoted by $\mathbf{g}^{-1}$, is a function from R_q to R^d which transforms an element $\mathbf{a} \in R_q$ into $\mathbf{A} \in R^d$, a vector of small polynomials such that $\mathbf{a} = \langle \mathbf{g}, \mathbf{A} \rangle \pmod q$. We integrate the RNS-friendly gadget decomposition from [7, 37].

3. METHODOLOGY and ALGORITHMS

HEAX targets a full-RNS variant of the CKKS scheme. RNS support was introduced in [16] and provided a performance boost. Three instances of the CKKS scheme are currently available, namely Microsoft SEAL [69], HEAAN [39], and HElib [41] libraries. We choose Microsoft SEAL because the other two are not fully based on RNS, i.e., have dependencies on multi-precision integer operations.

In this section, we summarize the algorithms in HEAX that implement the CKKS scheme's evaluation primitives as implemented in Microsoft SEAL [69]. Throughout the section, "`CKKS.`" is a prefix to CKKS specific methods. For example, symmetric-key encryption `SymEnc` is not prefixed since it is not specific to CKKS but rather a standard operation cryptosystems based on Ring Learning with Errors [54]. In text, methods are described without RNS or NTT forms for readability. In implementation and algorithms, polynomials are represented in RNS throughout evaluation and in NTT representation whenever possible. For completeness, we briefly cover the non-evaluation primitives of CKKS here. Our focus, evaluation primitives, are explained in subsections.

- `CKKS.Setup`(λ): For a security parameter λ, set a ring dimension n, a ciphertext modulus q, a special modulus p coprime to q, and a key distribution χ and an error distribution Ω over R.
- `SymEnc`$(\mathbf{m},\ \texttt{sk})$: Let $\mathbf{m} \in R$ be an input plaintext and $\texttt{sk} = \mathbf{s} \in R_{qp}$ be a secret key. Sample $\mathbf{a} \leftarrow U(R_{qp})$ and $\mathbf{e} \leftarrow \Omega$. Compute $\mathbf{b} = -\mathbf{a} \cdot \mathbf{s} + \mathbf{e} \in R_{qp}$. Return the ciphertext $\texttt{ct} = (\mathbf{c}_0,\ \mathbf{c}_1) = (\mathbf{b},\ \mathbf{a})$.
- `CKKS.KeyGen()`: Sample $\mathbf{s} \leftarrow \chi$. Return a secret key $\texttt{sk} = \mathbf{s}$ and a public key $\texttt{pk} = \texttt{SymEnc}(0,\ \texttt{sk})$.
- `CKKS.Enc`$(\mathbf{m},\ \texttt{pk})$: Let $\mathbf{m} \in R$ be an input plaintext and $\texttt{pk} = (\mathbf{b},\ \mathbf{a}) \in R_{qp}^2$ be a public key. Sample $\mathbf{u} \leftarrow \chi$ and $\mathbf{e}_0,\ \mathbf{e}_1 \leftarrow \Omega$. Compute $(\mathbf{c}_0',\ \mathbf{c}_1') = \mathbf{u} \cdot (\mathbf{b},\ \mathbf{a}) + (\mathbf{e}_0,\ \mathbf{e}_1) \pmod{qp}$. Return ciphertext $\texttt{ct} = (\mathbf{c}_0,\ \mathbf{c}_1) = (\mathbf{m},\ 0) + \lfloor (\mathbf{c}_0',\ \mathbf{c}_1')/p \rceil \in R_q^2$.
- `CKKS.Dec(ct, sk)`: Let $\texttt{ct} = (\mathbf{c}_0,\ \mathbf{c}_1) \in R_{q_\ell}^2$ be a ciphertext at the ℓ-th level,where $q_\ell = \prod_{i=0}^{\ell} p_i$, return $\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} \pmod{q_\ell}$.
- `KskGen`$(\texttt{sk}', \texttt{sk})$: Let $\texttt{sk} = \mathbf{s} \in R_{qp}$ be the generated secret key, $\texttt{sk}' = \mathbf{s}' \in R_{qp}$ be a different key, and a gadget vector $\mathbf{g} \in \mathbb{Z}^d$. Return a key switching key $\texttt{ksk} = (\mathbf{D}_0 \mid \mathbf{D}_1) \in R_{q_\ell p}^{(L+2)\times 2}$, where $(\mathbf{d}_{0,i},\ \mathbf{d}_{1,i}) \leftarrow \texttt{SymEnc}(g_i \cdot \mathbf{s}',\ \mathbf{s})$ for $i = 0,\ 1,\ \ldots,\ d-1$.
- `CKKS.RlkGen(sk)`: Let $\texttt{sk} = \mathbf{s} \in R_{qp}$ be the generated secret key, and a gadget vector $\mathbf{g} \in \mathbb{Z}^d$. Return a relinearization key $\texttt{rlk} \leftarrow \texttt{KskGen}(\mathbf{s}^2,\ \mathbf{s})$.
- `CKKS.GlkGen(sk, step)`: Let $\texttt{sk} = \mathbf{s} \in R_{qp}$ be the generated secret key, $\texttt{step} \in \mathbb{Z}$ be the number of steps that a ciphertext is supposed to be shifted (positive for left-handed shifting, negative for right-handed shifting), and a gadget vector $\mathbf{g} \in \mathbb{Z}^d$. Compute a key $\mathbf{s}' \in R_{qp}$ based on $\mathbf{s}$ and `step` (see [10] for details). Return a Galois (or rotation) key $\texttt{glk} \leftarrow \texttt{KskGen}(\mathbf{s}^2,\ \mathbf{s})$.

Algorithm 1 Standard Barrett Reduction

Input: $p < 2^w$, $x \in [0, (p-1)^2] \cap \mathbb{Z}$, and $u = \lfloor 2^{2w}/p \rfloor$
Output: $z \leftarrow x \pmod p$

$\alpha \leftarrow \lfloor \alpha / 2^{2w} \rfloor$ ▷ discard the lowest two words
$z_\varepsilon \leftarrow \alpha \cdot p$ ▷ double-word multiplication
$z \leftarrow x - z_\varepsilon$ ▷ double-word subtraction
if $z \geq p$ **then**
    $z \leftarrow z - p$
end if

Algorithm 2 Optimized Modular Multiplication

Input: $p < 2^{w-2}$, $x, y \in \mathbb{Z}_p$, and $y' = \lfloor y \cdot 2^w / p \rfloor$
Output: $z \leftarrow x \cdot y \pmod p$

$z \leftarrow x \cdot y \pmod{2^w}$ ▷ the lower word of the product
$t \leftarrow \lfloor x \cdot y' / 2^w \rfloor$ ▷ the upper word of the product
$z_\varepsilon \leftarrow t \cdot p \pmod{2^w}$ ▷ the lower word of the product
$z \leftarrow z - z_\varepsilon$ ▷ single-word subtraction
if $z \geq p$ **then**
    $z \leftarrow z - p$
end if

3.1. Number-Theoretic Transform (NTT)

Computing $\mathbf{c} = \mathbf{a} \cdot \mathbf{b} \in R_p$ is equivalent to computing the negacyclic convolution of their coefficient vectors in $\mathbb{Z}_p^n$: $c_j = \sum_{i=0}^{j} a_i b_{j-i} - \sum_{i=j+1}^{n-1} a_i b_{j-i+n} \pmod p$, $j = 0, 1, \ldots, n-1$. For a large n, which is at least 4096 in our implementation, it is asymptotically better to use the convolution theorem and perform a specific form of fast Fourier transform, i.e., NTT, over a finite field. Polynomials are kept in NTT form to reduce the number of NTT and its inverse (INTT) conversions. Fast NTT algorithms are well studied in lattice-based cryptography. We adapt the algorithms in [52] (Section 2.2) which analyzes fast NTT algorithms and introduces specific optimizations for negacyclic convolution. For a ring degree n, we choose a prime number $p = 1 \bmod 2n$ such that there exists a $2n$-th primitive root of unity ψ, i.e., $\psi^n = -1 \bmod p$.

There are optimizations to integer modular multiplication in the Microsoft SEAL library, which differ from those in [52].

- $\underline{\texttt{Mod}(x,\ p)}$: For a modulus p with at most w bits, given an integer $x \in [0,\ (p-1)^2]$, precompute $u = \lfloor 2^{2w}/p \rfloor$, and compute $z = x \pmod p$. We omit u from function inputs for simplicity. This follows the Barrett reduction algorithm in [8]. It can be used to reduce a single-word or double-word integer. $\texttt{Mod}(\mathbf{a}, p)$ performs $\texttt{Mod}(a_i,\ p)$ for all $i = 0, 1, \ldots, n-1$.
- $\underline{\texttt{MulRed}(x,\ y,\ y',\ p)}$: For w-bit words and a modulus $p < 2^{w-2}$, given $x,\ y \in \mathbb{Z}_p$ and precomputed $y' = \lfloor y \cdot 2^w / p \rfloor$, compute $x \cdot y \pmod p$ according to Algorithm 2. MulRed optimizes modular multiplications when one of the operands is a known constant, e.g., a power of the root of unity, that can be precomputed. Compared to Mod, this algorithm has less multi-word operations, hence, it is faster.

The value of w is chosen as 54 in HEAX as opposed to 64 in Microsoft SEAL. We explain the reason in more detail in Section 4. Modulus p has at most 52 bits.

- $\underline{\texttt{NTT}_p(\mathbf{a})}$: Given $\mathbf{a} \in \mathbb{Z}_p^n$, compute $\tilde{\mathbf{a}} \in \mathbb{Z}_p^n$ such that $\tilde{a}_j = \sum_{i=0}^{n-1} a_i \psi^{(2i+1)j}$, according to Algorithm 3.
- $\underline{\texttt{INTT}_p(\tilde{\mathbf{a}})}$: Given $\tilde{\mathbf{a}} \in \mathbb{Z}_p^n$, compute $\mathbf{a} \in \mathbb{Z}_p^n$ such that $a_j = \frac{1}{n} \sum_{i=0}^{n-1} \tilde{a}_i \psi^{-(2i+1)j}$, according to Algorithm 4.

Algorithm 3 NTT

Input: $\mathbf{a} \in \mathbb{Z}_p^n$, $p \equiv 1 \mod 2n$, $\mathbf{Y} \in \mathbb{Z}_p^n$ storing powers of ψ in bit-reverse order, and $\mathbf{Y}' = \lfloor \mathbf{Y} \cdot 2^w / p \rfloor$.
Output: : $\tilde{\mathbf{a}} \leftarrow \texttt{NTT}_p(\mathbf{a})$ in bit-reverse ordering.

for $(m = 1;\ m < n;\ m = 2m)$ **do**
    for $(i = 0;\ i < m;\ i{+}{+})$ **do**
        for $(j = \frac{i \cdot n}{m};\ j < \frac{(2i+1)n}{2m};\ j{+}{+})$ **do**
            $v = \texttt{MulRed}(a_{j+\frac{n}{m}},\ y_{m+i},\ y'_{m+i},\ p)$
            $a_{j+\frac{n}{m}} = a_j - v \pmod p$
            $a_j = a_j + v \pmod p$
        end for
    end for
end for
$\tilde{\mathbf{a}} \leftarrow \mathbf{a}$

Algorithm 4 INTT

Input: $\tilde{\mathbf{a}} \in \mathbb{Z}_p^n$, $p \equiv 1 \mod 2n$, $\mathbf{Y} \in \mathbb{Z}_p^n$ storing powers of ψ^{-1} divided by 2 in bit-reverse order, and $\mathbf{Y}' = \lfloor \mathbf{Y} \cdot 2^w / p \rfloor$.
Output: : $\mathbf{a} \leftarrow \texttt{INTT}_p(\tilde{\mathbf{a}})$ in bit-reverse ordering.

for $(m = n/2;\ m \geq 1;\ m = m/2)$ **do**
    for $(i = 0;\ i < m;\ i{+}{+})$ **do**
        for $(j = \frac{i \cdot n}{m};\ j < \frac{(2i+1)n}{2m};\ j{+}{+})$ **do**
            $v = \tilde{a}_j - \tilde{a}_{j+\frac{n}{m}} \pmod p$
            $\tilde{a}_j = (\tilde{a}_j + \tilde{a}_{j+\frac{n}{m}})/2 \pmod p$
            $\tilde{a}_{j+\frac{n}{m}} = \texttt{MulRed}(v,\ y_{m+i},\ y'_{m+i},\ p)$
        end for
    end for
end for
$\mathbf{a} \leftarrow \tilde{\mathbf{a}}$

3.2. Addition and Multiplication

These two operations are performed in RNS and NTT form:

- $\underline{\texttt{CKKS.Add}(\texttt{ct}_0,\ \texttt{ct}_1)}$: Given ciphertexts $\texttt{ct}_0,\ \texttt{ct}_1 \in R_{q_\ell}^2$ encrypting $\texttt{pt}_0,\ \texttt{pt}_1 \in R$, generate $\texttt{ct}' = \texttt{ct}_0 + \texttt{ct}_1 \in R_{q_\ell}^2$ which is equivalent to the encryption of $\texttt{pt}_0 + \texttt{pt}_1 \in R$.
- $\underline{\texttt{CKKS.Mul}(\texttt{ct}_0,\ \texttt{ct}_1)}$: Given ciphertexts $\texttt{ct}_0,\ \texttt{ct}_1 \in R_{q_\ell}^2$ encrypting $\texttt{pt}_0,\ \texttt{pt}_0 \in R$, generate $\texttt{ct}' \in R_{q_\ell}^3$ according to Algorithm 5 which encrypts $\texttt{pt}_0 \cdot \texttt{pt}_1 \in R$.

3.3. Rescaling

In CKKS, a plaintext is the encoding of message slots multiplied by a scale Δ. Note that by multiplying plaintexts, ciphertexts, or a plaintext and a ciphertext, their scales are also multiplied. Therefore, the scale grows exponentially on the number of multiplications. It will quickly grow larger than q_L, which causes decryption failure. Besides, adding two operands with different scales produces an incorrect result. Rescaling

Algorithm 5 Ciphertext Multiplication

Input: $\texttt{ct}_0 = (\tilde{\mathbf{A}}_0,\ \tilde{\mathbf{A}}_1),\ \texttt{ct}_1 = (\tilde{\mathbf{B}}_0,\ \tilde{\mathbf{B}}_1) \in (\prod_{i=0}^{\ell} R_{p_i})^2$
Output: $\texttt{ct} = (\tilde{\mathbf{C}}_0,\ \tilde{\mathbf{C}}_1,\ \tilde{\mathbf{C}}_2) \in (\prod_{i=0}^{\ell} R_{p_i})^3$

1: **for** $(i = 0;\ i \le \ell;\ i = i+1)$ **do**
2: $\quad \tilde{\mathbf{c}}_{0,i} \leftarrow \texttt{Mod}(\tilde{\mathbf{a}}_{0,i} \odot \tilde{\mathbf{b}}_{0,i},\ p_i)$
3: $\quad \tilde{\mathbf{c}}_{1,i} \leftarrow \texttt{Mod}(\tilde{\mathbf{a}}_{0,i} \odot \tilde{\mathbf{b}}_{1,i} + \tilde{\mathbf{a}}_{1,i} \odot \tilde{\mathbf{b}}_{0,i},\ p_i)$
4: $\quad \tilde{\mathbf{c}}_{2,i} \leftarrow \texttt{Mod}(\tilde{\mathbf{a}}_{1,i} \odot \tilde{\mathbf{b}}_{1,i},\ p_i)$
5: **end for**

Algorithm 6 RNS Flooring

Input: $\tilde{\mathbf{C}} = (\tilde{\mathbf{c}}_0,\ \ldots,\ \tilde{\mathbf{c}}_{\ell+1}) \in \mathbb{Z}_{p_0}^n \times \cdots \times \mathbb{Z}_{p_\ell}^n \times \mathbb{Z}_p^n$.
Output: $\tilde{\mathbf{C}}' = (\tilde{\mathbf{c}}'_0,\ \ldots,\ \tilde{\mathbf{c}}'_\ell) \in \mathbb{Z}_{p_0}^n \times \cdots \times \mathbb{Z}_{p_\ell}^n$.

1: $\mathbf{a} \leftarrow \texttt{INTT}_p(\tilde{\mathbf{c}}_{\ell+1})$
2: **for** $(i = 0;\ i \le \ell;\ i = i+1)$ **do**
3: $\quad \bar{\mathbf{r}} \leftarrow \texttt{Mod}(\mathbf{a},\ p_i)$
4: $\quad \tilde{\mathbf{r}} \leftarrow \texttt{NTT}_{p_i}(\bar{\mathbf{r}})$
5: $\quad \tilde{\mathbf{c}}'_i \leftarrow \tilde{\mathbf{c}}_i - \tilde{\mathbf{r}} \pmod{p_i}$
6: $\quad \tilde{\mathbf{c}}'_i \leftarrow \texttt{Mod}\left(\left[p^{-1}\right]_{p_i} \cdot \tilde{\mathbf{c}}'_i,\ p_i\right)$
7: **end for**

changes (mostly reduces) the scale of a ciphertext. In order to support full-RNS, rescaling changes the scale by a set of *fixed* ratios, i.e., one of the RNS prime numbers.

• $\texttt{Floor}(\tilde{\mathbf{C}}, p)$: Given $\tilde{\mathbf{C}}$, the RNS and NTT form of $\mathbf{c} \in R_{q_\ell p}$, generate $\tilde{\mathbf{C}}'$, the RNS and NTT form of $\mathbf{c}' = \lfloor p^{-1} \cdot \mathbf{c} \rceil \in R_{q_\ell}$ according to Algorithm 6.

• $\texttt{CKKS.Rescale(ct)}$: Given a ciphertext $\texttt{ct} = (\tilde{\mathbf{C}}_0,\ \tilde{\mathbf{C}}_1) \in R_{q_\ell}^2$ with scale Δ, generate a new ciphertext $\texttt{ct}' = (\texttt{Floor}(\tilde{\mathbf{C}}_0),\ \texttt{Floor}(\tilde{\mathbf{C}}_1)) \in R_{q_{\ell-1}}^2$ with scale Δ/p_ℓ in RNS and NTT form.

3.4. Key Switching

Key switching is a technique to make a ciphertext decryptable with a different secret key homomorphically. Various gadget decomposition methods can be adopted to balance noise growth and execution time. Given q_{d-1}, the product of coprime integers $p_0,\ \ldots,\ p_{d-1}$, and q_ℓ divides q_{d-1}, define gadget decomposition $R_{q_\ell} \mapsto R^d$ as $\mathbf{g}^{-1}(\mathbf{a}) = \left([\mathbf{a}]_{p_i}\right)_{0 \le i \le d-1}$, and gadget vector as $\mathbf{g} = \left(\pi_i \left[\pi_i^{-1}\right]_{p_i}\right)_{0 \le i \le d-1}$ where $\pi_i = \frac{q_{d-1}}{p_i}$. This choice of gadget decomposition contributes to a fast key switching and high noise growth. With the special modulus p and a rescaling at the end of key switching, explained in [15], key switching is almost noise-free.

• $\texttt{KeySwitch(ct, ksk)}$: Given a ciphertext $\texttt{ct} = (\mathbf{c}_0, \mathbf{c}_1) \in R_{q_\ell}^2$ decryptable with secret key $\mathbf{s}$ and a key switching key $\texttt{ksk} = (\mathbf{D}_0 \mid \mathbf{D}_1) \in R_{q_\ell p}^{(L+2)\times 2}$, where $\mid$ appends one column vector to another, generate a new ciphertext $\texttt{ct}' = (\mathbf{c}'_0,\ \mathbf{c}'_1) \in R_{q_\ell}^2$ decryptable with secret key $\mathbf{s}'$ as follows:

1. $\mathbf{A} \leftarrow \mathbf{g}_\ell^{-1}(\mathbf{c}_1) = \left([\mathbf{c}_1]_{p_0},\ \ldots,\ [\mathbf{c}_1]_{p_\ell}\right) \in R^\ell$.
2. $(\mathbf{c}''_0,\ \mathbf{c}''_1) \leftarrow (\langle \mathbf{A},\ \mathbf{D}_0 \rangle,\ \langle \mathbf{A},\ \mathbf{D}_1 \rangle) \in R_{q_\ell p}^2$.

Algorithm 7 Key Switching

Input: $\texttt{ct} = (\tilde{\mathbf{C}}_0,\ \tilde{\mathbf{C}}_1) \in (\prod_{i=0}^{\ell} R_{p_i})^2$, and $\texttt{ksk} = \left(\left(\tilde{\mathbf{D}}_{i,0}\right)_{0 \le i \le L+1} \mid \left(\tilde{\mathbf{D}}_{i,1}\right)_{0 \le i \le L+1}\right) \in (p \prod_{i=0}^{L} R_{p_i})^{(L+2)\times 2}$
Output: $\texttt{ct}' = (\tilde{\mathbf{C}}'_0,\ \tilde{\mathbf{C}}'_1) \in (\prod_{i=0}^{\ell} R_{p_i})^2$

1: a zero ciphertext $\texttt{ct}'' = (\tilde{\mathbf{C}}''_0,\ \tilde{\mathbf{C}}''_1) \in (p \prod_{i=0}^{\ell} R_{p_i})^2$
2: **for** $(i = 0;\ i \le \ell;\ i = i+1)$ **do**
3: $\quad \bar{\mathbf{a}} \leftarrow \texttt{INTT}_{p_i}(\tilde{\mathbf{c}}_{1,i})$
4: $\quad$ **for** $(j = 0;\ j \le \ell;\ j = j+1)$ **do**
5: $\quad\quad$ **if** $i \ne j$ **then**
6: $\quad\quad\quad \bar{\mathbf{b}} \leftarrow \texttt{Mod}(\bar{\mathbf{a}},\ p_j)$
7: $\quad\quad\quad \tilde{\mathbf{b}} \leftarrow \texttt{NTT}_{p_j}(\bar{\mathbf{b}})$
8: $\quad\quad$ **else**
9: $\quad\quad\quad \tilde{\mathbf{b}} \leftarrow \tilde{\mathbf{a}}$
10: $\quad\quad$ **end if**
11: $\quad\quad \tilde{\mathbf{c}}''_{0,j} \leftarrow \tilde{\mathbf{c}}''_{0,j} + \tilde{\mathbf{b}} \odot \tilde{\mathbf{d}}_{i,0,j} \pmod{p_j}$
12: $\quad\quad \tilde{\mathbf{c}}''_{1,j} \leftarrow \tilde{\mathbf{c}}''_{1,j} + \tilde{\mathbf{b}} \odot \tilde{\mathbf{d}}_{i,1,j} \pmod{p_j}$
13: $\quad$ **end for**
14: $\quad \bar{\mathbf{b}} \leftarrow \texttt{Mod}(\bar{\mathbf{a}}\, p)$
15: $\quad \tilde{\mathbf{b}} \leftarrow \texttt{NTT}_p(\bar{\mathbf{b}})$
16: $\quad \tilde{\mathbf{c}}''_{0,\ell+1} \leftarrow \tilde{\mathbf{c}}''_{0,\ell+1} + \tilde{\mathbf{b}} \odot \tilde{\mathbf{d}}_{0,i,L+1} \pmod{p_j}$
17: $\quad \tilde{\mathbf{c}}''_{1,\ell+1} \leftarrow \tilde{\mathbf{c}}''_{1,\ell+1} + \tilde{\mathbf{b}} \odot \tilde{\mathbf{d}}_{1,i,L+1} \pmod{p_j}$
18: **end for**
19: $\texttt{ct}' \leftarrow (\texttt{Floor}(\tilde{\mathbf{C}}''_0, p),\ \texttt{Floor}(\tilde{\mathbf{C}}''_1, p))$
20: $\texttt{ct}' \leftarrow \texttt{CKKS.Add}(\texttt{ct},\ \texttt{ct}')$

3. $\texttt{ct}' = (\mathbf{c}'_0, \mathbf{c}'_1) \leftarrow (\texttt{Floor}(\mathbf{c}''_0, p), \texttt{Floor}(\mathbf{c}''_1, p)) \in R_{q_\ell}^2$.
4. $\texttt{ct}' \leftarrow \texttt{CKKS.Add}(\texttt{ct}', \texttt{ct})$.

The input and output ciphertext and the key switching key are in RNS representation and in the NTT form. Algorithm 7 describes the full-RNS variant.

Relinearization. Multiplying two ciphertexts generates a ciphertext $\texttt{ct}' \in R_{q_\ell}^3$ with three components (Algorithm 5). $\texttt{ct}'$ can be decrypted as $\langle \texttt{ct}',\ (1,\ \mathbf{s},\ \mathbf{s}^2) \rangle$ in R_{q_ℓ} where $\mathbf{s}$ is the secret key. To stop ciphertexts from expanding, we *relinearize* the ciphertext to transform it back to two components. Relinearization is fundamentally a key-switching operation. It transforms a ciphertext decryptable with $\mathbf{s}^2$ to a new ciphertext decryptable with $\mathbf{s}$ using relinearization key.

• $\texttt{CKKS.Relin(ct, rk)}$: Given a ciphertext $(\tilde{\mathbf{c}}_0,\ \tilde{\mathbf{c}}_1,\ \tilde{\mathbf{c}}_2) \in (\prod_{i=0}^{\ell} R_{p_i})^3$ and a relinearization key $\texttt{rk}$ (can be seen as $\texttt{ksk}$), generates a ciphertext $(\tilde{\mathbf{c}}'_0,\ \tilde{\mathbf{c}}'_1) \in (\prod_{i=0}^{\ell} R_{p_i})^2$ as follows:

1. Compute $\texttt{ct}^* = \texttt{KeySwitch}(\tilde{\mathbf{c}}_2,\ \texttt{rk}) \in R_{q-1}^2$.
2. $\tilde{\mathbf{c}}'_0 \leftarrow \tilde{\mathbf{c}}_0 + \tilde{\mathbf{c}}^*_0$, $\tilde{\mathbf{c}}'_1 \leftarrow \tilde{\mathbf{c}}_1 + \tilde{\mathbf{c}}^*_1$.

Rotation. A plaintext is the encoding of a vector of messages that are evaluated independently. Rotation is applied on a ciphertext to move messages across slots. Each rotation pattern has a corresponding secret key $\mathbf{s}'$ derived from the original secret key $\mathbf{s}$. Rotation generates a ciphertext that after being decrypted with $\mathbf{s}'$ is the encoding of the rotated message vector. The single costly part of rotation is also $\texttt{KeySwitch}$. We refer readers to [17] and [10] for more details.

4. ARCHITECTURE

In this section, we describe our proposed architectures for NTT/INTT, Multiplication, and Key Switching modules.

Word Size and Native Operations. Microsoft SEAL library [69] is developed for x86 architectures with 64-bit native operations. However, on FPGAs, the bit-width of Digital Signal Processing (DSP) units that perform multiplication may vary, and hence, it is more efficient to have a different bit-width for native operations. For example, the two FPGA chips that we have implemented our architecture on have 27-bit DSP units. Choosing 27 or 54-bit multiplications as our native operation enables us to use less DSP units to do the same computation. Naive construction of a 64-bit multiplier requires nine 27-bit DSPs. Whereas, a 54-bit multiplier requires only four. However, by reducing the bit-width of the RNS components, one may need to increase the number of such components; roughly speaking, by a factor of $\frac{64}{54} \approx 1.2$.

In practice, small ciphertext moduli are usually less than 54 bits and thus, we do not need to increase the number of moduli. It is worth-mentioning that leveraging more sophisticated multi-word multiplication algorithms such as Toom-Cook, one can implement 64-bit multiplication using *five* 27-bit multipliers together with more bit-level and Addition operations. Overall, by switching from 64-bit native operations to 54-bit, we observe between $1.4\times$ to $2.25\times$ reduction in the number of DSP units needed (depending on the HE parameters). Please note that with 54-bit word size, we need to make sure all of the ciphertext moduli (p_i) are (i) less than 52-bit to ensure the correctness of Algorithm 2 and (ii) congruent to 1 mod $2n$ to support NTT as described in Section 3.1. We have precomputed all of such moduli for different parameters.

4.1. MULT Module

We first describe the architecture of the MULT module which can process both ciphertext-ciphertext (C-C) as well as ciphertext-plaintext (C-P) homomorphic multiplications. In what follows, we discuss the architecture for C-C multiplication as C-P is a special case of C-C. In Microsoft SEAL, to achieve superior performance, ciphertexts and plaintexts are in NTT form by default and are converted back to the original form only if needed. Therefore, homomorphic multiplication is simply a series of dyadic multiplications on different components of two input ciphertexts as described in Algorithm 5.

The MULT module, depicted in Figure 1, encompasses nc_{DYD}-many *Dyadic Cores* (*nc* stands for number of cores). Therefore, MULT module can compute nc_{DYD} dyadic multiplication at each clock cycle. Each Dyadic core takes as input two polynomial coefficients (Op_1 and Op_2), two precomputed constant values (R_1 and R_2), and one-word prime p and outputs the multiplication result.

In a general case, each ciphertext can have more than two components. For example, consider a scenario where $\texttt{ct}_1$ (or $\texttt{ct}_2$) is the output of a previous C-C Mult (not relinearized) and now has three components. Let us denote the number of components in $\texttt{ct}_1$ and $\texttt{ct}_2$ by α and β, respectively. The outcome of homomorphic multiplication is a ciphertext with $\alpha+\beta-1$ components. Each ciphertext component is represented in a RNS form. Recall that in homomorphic multiplication (Algorithm 5), the computation can be carried out independently on each RNS basis of two ciphertexts and we leverage this property to reduce BRAM utilization. Minimum BRAM utilization is achieved by storing one residue of each ciphertext component. However, this approach significantly increases data transfer from CPU to FPGA from $(\alpha+\beta)\cdot n$ words to $(\alpha\cdot\beta+\min(\alpha,\beta))\cdot n$ words since we need to compute all pairwise combinations of $\texttt{ct}_1$ and $\texttt{ct}_2$ components. Thus, we allocate α-many memories of size n for $\texttt{ct}_1$ and β-many memories for $\texttt{ct}_2$ to hold *one* residue of *all* ciphertext components. As a result, we achieve $\mathcal{O}\big((\alpha+\beta)\cdot n\big)$ data transfer and BRAM consumption.

In order to fully utilize all nc_{DYD} Dyadic cores – regardless of the values of α and β – we read nc_{DYD} coefficients from one of the polynomials of $\texttt{ct}_1$ and $\texttt{ct}_2$ at every clock cycle. However, each unit of on-chip memory, i.e., Block RAMs (BRAM), only supports one read and one write at each clock cycle. In order to read many coefficients from one polynomial at each cycle, we store each polynomial across nc_{DYD}-many parallel memory blocks that share common read/write address signals as depicted in Figure 1. Let us call the aggregation of one row among different BRAMs as a *memory element* (ME). Therefore, at every cycle, one memory element (ME1/ME2) is read from $\texttt{ct}_1/\texttt{ct}_2$ memory banks and the result (ME3) is written to a separate output memory.

4.2. NTT Module

NTT/INTT is the most computationally intensive low-level operation in homomorphic encryption schemes. In what follows, we use NTT to refer to both NTT and INTT modules. At the end of this section, we discuss the differences between these two modules. As can be seen from Algorithm 7, in `KeySwitch`, NTT/INTT is computed many times on different intermediate polynomials. However, the number of times that we need to compute NTT/INTT is different in distinct parts of the Algorithm. In order to have a fully pipelined architecture, we allocate one NTT module per each NTT operation in Algorithm 7. The relative throughput-rate among different NTT instances also depends on the selected FHE parameters, which is application-dependent. As a result, we need to have a generic architecture for NTT module such that the throughput can be adjusted as needed. This, in turn, is translated to the number of NTT *cores* that is dedicated to a given NTT module. Figure 3 shows the internal architecture of a single NTT core which accepts two coefficients, one twiddle factor, one precomputed value, and a prime as inputs and computes two transformed coefficients as the outputs. From the functionality perspective, the architecture follows Algorithms 3 and 4.

The throughput of the NTT module is proportional to the

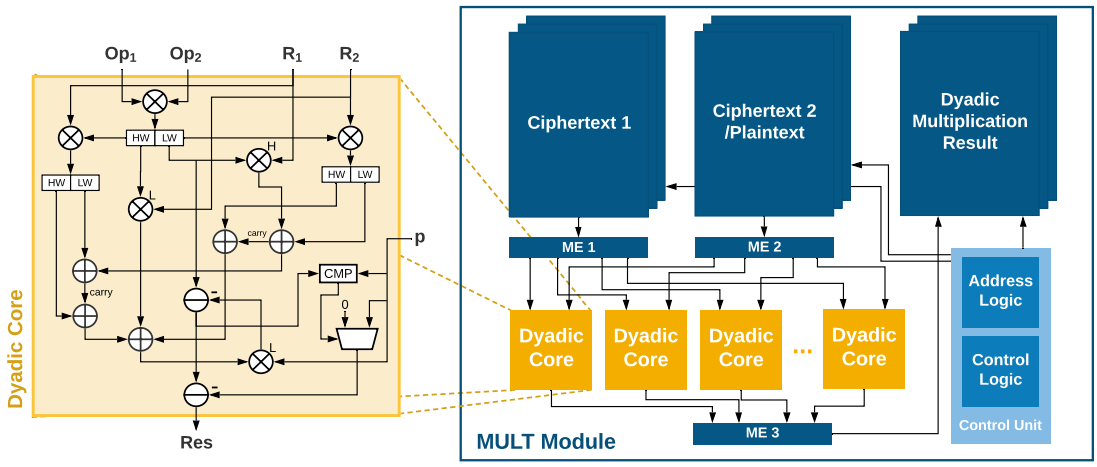

Figure 1: Architecture of MULT module.

number of NTT cores that it encompasses. We denote the number of NTT cores as nc_{NTT}. Ideally at each clock cycle, and given full utilization of NTT cores, $2nc_{\text{NTT}}$ coefficients are transformed. Similar to MULT module, we store each polynomial across many parallel BRAMs that share common read/write address signals as depicted in Figure 3. This is possible thanks to the *aligned* access pattern in NTT: while access pattern changes during NTT, the number of consecutive accesses to the polynomial is always a power of two.

Access Pattern. At high-level, the NTT module computes NTT of a polynomial of size n in $\log n$ *stages*. In each stage, the module computes the transformed result of $2nc_{\text{NTT}}$ coefficients, thus, requiring $\frac{n}{2nc_{\text{NTT}}}$ *steps* to finish one stage. However, the access pattern changes from one stage to another. Figure 2 illustrates the two types of access patterns in NTT. During the first $\log n - \log nc_{\text{NTT}} - 1$ stages, these pairs of coefficients are stored in different MEs. Let us call these *Type 1* stages. For instance, consider $n = 4096$ and $nc_{\text{NTT}} = 8$, during the first step of the first stage of NTT, $x[0]$ (in ME_0) and $x[2048]$ (in ME_{256}) should be passed to the first NTT core. More precisely, polynomial coefficient $x[j]$ ($j = 0, 1, \ldots, \frac{n}{2} - 1$) is passed together with $x[j + \frac{n}{2}]$ to a given NTT core. In general, during i^{th} stage, $x[j + m]$ ($j = 0, 1, \ldots, \frac{n}{2^{1+i}} - 1$) is passed along with $x[j + m + \frac{n}{2^{1+i}}]$ where $m \in \{\frac{h \cdot n}{2^i} \,|\, h = 0, 1, \ldots, i\}$. The address of the ME that is fetched at stage i and step j in Type 1 stages is computed in *Address Logic* as follows:

$$Addr\{\text{ME}_{\text{coeff}}\}_{i,j} = ((j >> 1)\,\&\,(2^s - 1)) + ((j >> (s+1)) << (s+2)) + s \cdot (j \bmod 2)$$

where $s = \log n - \log nc_{\text{NTT}} - 2 - i$, "&" is bit-wise AND operation, $>>$ denotes right-shift, and $<<$ is left-shift.

As soon as $\frac{n}{2^i} = 2nc_{\text{NTT}}$, the inter-ME data dependency no longer exists, and pairs of coefficients are selected from within each ME independently, i.e., *Type 2* stages.

In Type 1 stages, coefficients within two fetched MEs are always accessed in the same order. For example, the second coefficient in each ME is always passed to the second NTT core. However, in Type 2 stages, a coefficient at specific location of ME is passed to a different NTT core or even different inputs of an NTT core ($c_{\text{in.a}}$ or $c_{\text{in.b}}$) during consecutive stages. Therefore, coefficients have to be *reordered* to be passed to NTT cores. Later in this section, we will discuss an efficient method for reordering the coefficients. After the computations in NTT cores, coefficients have to be reordered again and stored in the same positions as they were accessed.

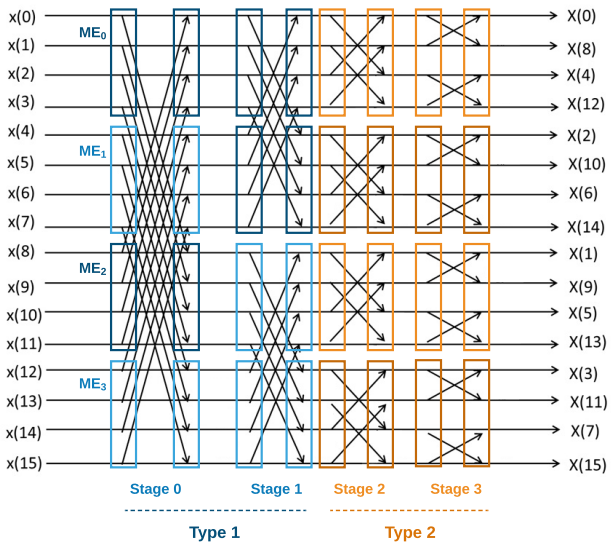

Figure 2: Access pattern of Type 1 and Type 2 stages in NTT.

The access pattern for twiddle factors, i.e., $\mathbf{Y}$ and $\mathbf{Y}'$ in Algorithm 3, is different from the access pattern of coefficients. At stage i, only 2^i *unique* values of twiddle factors, starting at index 2^i of twiddle polynomial, are used. Since in the worst-case scenario, nc_{NTT} unique twiddle factors are used in a single *step* of NTT, we store twiddle factors in batches of size nc_{NTT} in parallel as shown in Figure 3 (note that twiddle factors' MEs are half size of polynomial coefficients' MEs). As a result, we can divide the access pattern of twiddle factors into four

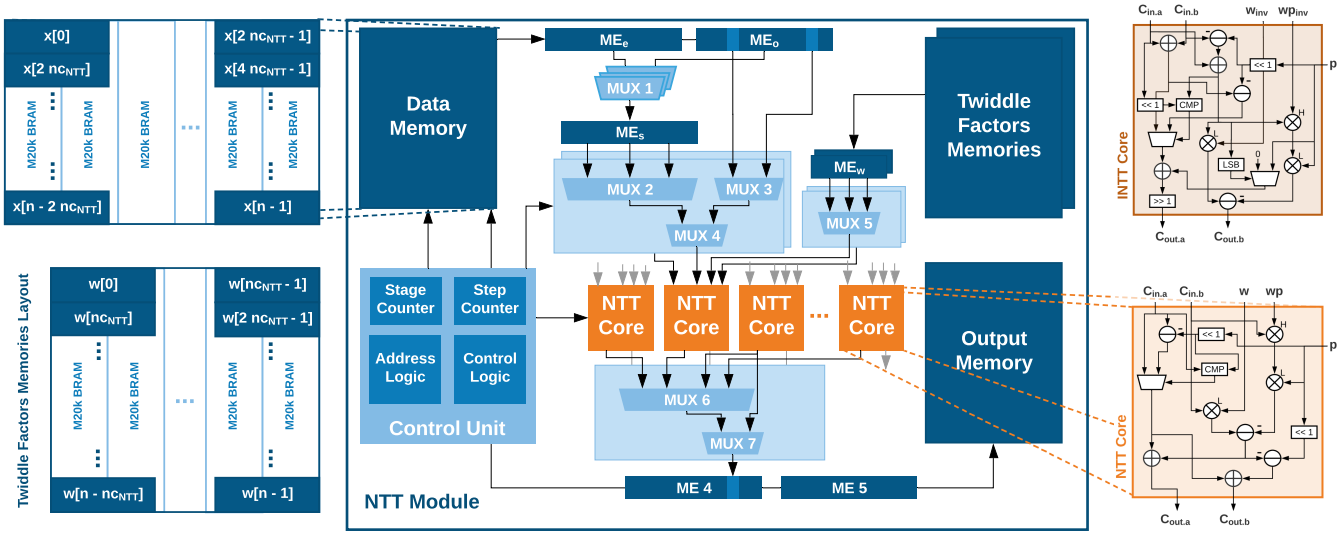

Figure 3: Architecture of NTT module.

groups: (i) in the first group of stages where $2^i < nc_{\text{NTT}}$, only ME_0 (first ME) is accessed throughout the stage computation and one (or more) twiddle factor(s) is(are) *broadcasted* into different NTT cores. (ii) At stage $\log nc_{\text{NTT}}$, only ME_1 is accessed but no broadcast is needed as each twiddle factor inside ME is passed to a separate NTT core. (iii) During stages where $\log nc_{\text{NTT}} < i < \log n - 1$, $2^{i-\log nc_{\text{NTT}}}$ unique MEs are fetched and passed to NTT cores, and finally, (iv) at stage $\log n - 1$, a new ME of twiddle factors is read from BRAM at every step. The address of the twiddle factor ME that is fetched at stage i and step j is computed in Address Logic as follows:

$$\text{Addr}\{\text{ME}_\text{w}\}_{i,j} = (j >> (\log n - 1 - i)) + (1 << (j - \log nc_{\text{NTT}}))$$

Reordering Coefficients and Customized Multiplexers. During Type 1 stages, once the ME is fetched, passing each coefficient in ME to the right NTT core (and right input wire) is straightforward and it can be summarized as follows:

$$\begin{cases} \text{c}^{\ell}_{\text{in.a}} = \text{ME}_e[\ell + (j \bmod 2) \cdot nc_{\text{NTT}}] \\ \text{c}^{\ell}_{\text{in.b}} = \text{ME}_o[\ell + (j \bmod 2) \cdot nc_{\text{NTT}}] \end{cases}$$

where $\text{c}^{\ell}_{\text{in.a}}$ (respectively $\text{c}^{\ell}_{\text{in.b}}$) is the input coefficient a (respectively b) of ℓ^{th} NTT core, ME_e (respectively ME_o) is the memory element at "even" ("odd") read cycles, i.e., $j \bmod 2 = 0$ ($j \bmod 2 = 1$) where j is the step number. In Type 2 stages, $\text{c}^{\ell}_{\text{in.a}}$ (or $\text{c}^{\ell}_{\text{in.b}}$) should receive data from one of the coefficients in ME depending on the value of ℓ and i. The naive approach is to use one multiplexer (MUX) per each coefficient input of every NTT core that selects one number from $2nc_{\text{NTT}}$ fetched numbers. We denote such multiplexer as $\text{MUX}^{2nc_{\text{NTT}}}$. As a result, we need $2nc_{\text{NTT}}$-many $\text{MUX}^{2nc_{\text{NTT}}}$ to pass coefficients to NTT cores and the same number of MUXs to reorder them to be written back to the memory. These MUXs not only make the process of placement and route more challenging but also consume enormous number of registers and logic blocks. Moreover, scaling the NTT module to higher number of cores (> 32) is inefficient due to super-linear resource consumption with respect to nc_{NTT}. In many cases, synthesis tools failed to place and route the required resources to realize these MUXs. In contrast, we take advantage of the fact that *NTT cores' inputs have a different number of possibilities from which they select the correct coefficient at a given stage*. For example, during Type 2 stages, $\text{c}^{0}_{\text{in.a}}$ only receives coefficients from first word of ME_s regardless of the stage or step number.

In the worst-case scenario, there are $\log nc_{\text{NTT}}$ different indices from which a coefficient should be accessed from ME_s for a particular NTT core input. Therefore, instead of using $(4 \cdot nc_{\text{NTT}})$-many $\text{MUX}^{2nc_{\text{NTT}}}$, we instantiate $(4 \cdot nc_{\text{NTT}})$-many MUXs of size at most $\text{MUX}^{\log 2nc_{\text{NTT}}}$. The selection signal of these MUXs is set to $s = \log n - 1 - i$ (i being the stage number). The corresponding inputs ($\text{MUX}\{\text{c}^{\ell}_{\text{in.a}}\}(\alpha)$ and $\text{MUX}\{\text{c}^{\ell}_{\text{in.b}}\}(\alpha)$) from which a coefficient should be selected are assigned based on the following formula:

$$= \begin{cases} \text{ME}_s[(\ell \,\&\, (2^s - 1)) + ((i >> s) << (s + \ell))] \\ \text{ME}_s[(\ell \,\&\, (2^s - 1)) + ((i >> s) << (s + \ell)) + 2^s] \end{cases}$$

where $\text{MUX}\{\text{c}^{\ell}_{\text{in.a}}\}(\alpha)$ is the α-th input wire of the MUX that selects the corresponding input coefficient from ME_s for ℓ-th NTT core, thus, $0 \leq \alpha < \log nc_{\text{NTT}}$. Creating customized multiplexers for twiddle factors is significantly simpler. As we discussed previously, during group i stages, only ME_0 is accessed and depending on the stage number, one or more of its twiddle factors are selected and broadcasted to different NTT cores. Therefore, from the perspective of each $\text{w}^{\ell}_{\text{in}}$ (or $\text{wp}^{\ell}_{\text{in}}$), there are $\log nc_{\text{NTT}}$ number of possibilities from which the twiddle factor can be selected. This is implemented as $\text{MUX}^{\log 2nc_{\text{NTT}}}$. During group ii-iv stages, $\text{w}^{\ell}_{\text{in}}$ (respectively $\text{wp}^{\ell}_{\text{in}}$) is selected as $\text{ME}_w[\ell]$ (respectively $\text{ME}_{wp}[\ell]$).

Two-Stage Read, Compute, and Write. Storing polynomial coefficients across different memory blocks solves simultaneous memory accesses. However, the NTT cores cannot be fully utilized at all times due to the following reason. During Type 1 stages, coefficients that should be passed to each NTT core are not located in the same memory element. Therefore, two different memory elements should be read (Read$^{\alpha}1$ and Read$^{\alpha}2$) before the computation (Comp$^{\alpha}$) can start which introduces 50% bubble in the NTT core pipeline (please see Figure 4). More precisely, first $\log n - \log nc_{\text{NTT}} - 1$ stages face this problem. Given that NTT modules consume most of the FPGA resources, this in turn means, the throughput of the entire design will be reduced to $(\log n - \log nc_{\text{NTT}} - 1)/\log n$.

In order to solve this problem, we propose to double the size of MEs and store twice as many consecutive coefficients, i.e., $2nc_{\text{NTT}}$, in each memory element. Meanwhile, we reduce the depth of the memories that store the polynomial by half. The modified pipeline is shown in Figure 4. Even though it is still necessary to read two MEs (during two consecutive cycles) before starting the computation, we can now transform *two* MEs in the next two cycles (Comp$^{\alpha}1$ and Comp$^{\alpha}2$) and store them back in the memory (Write$^{\alpha}1$ and Write$^{\alpha}2$). Our solution does not reduce the delay of the computation but it provides us the full utilization of NTT core during time. As we will discuss in Section 6, BRAM is the most constrained resource. In order to have minimal BRAM usage, all of the reads and writes during different NTT stages are *inplace*, and no additional BRAM is used to store intermediate values.

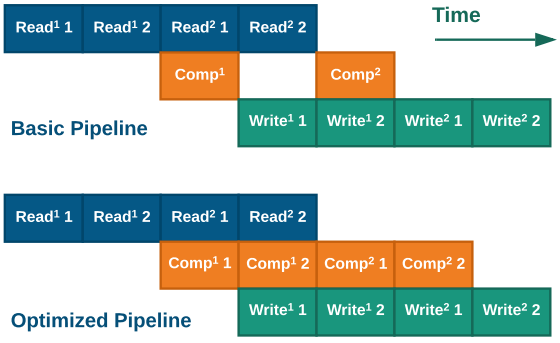

Figure 4: High-level pipeline of NTT module.

Putting it all together. Figure 3 illustrates the full architecture of NTT module. On the three corners exist data memory, twiddle factors' memories, and output memory. Based on the value of *step and stage counters*, the corresponding addresses of the MEs for data and twiddle factor memories are generated. At every cycle, one ME is fetched from data memory and is stored in ME$_e$ and ME$_o$ registers every other cycles, respectively. For each input coefficient of NTT cores, i.e., $c^{\ell}_{\text{in.a}}$ and $c^{\ell}_{\text{in.b}}$, a set of multiplexers select the correct coefficient from ME$_e$ and ME$_o$ (depicted as light blue boxes in Figure 3).

During Type 1 stages, $c^{\ell}_{\text{in.a}}$ is selected from one of the two positions from ME$_e$ and $c^{\ell}_{\text{in.b}}$ from ME$_o$, respectively. This selection is performed by MUX3. During Type 2 stages, first one of the ME$_e$ or ME$_o$ is selected using $2nc_{\text{NTT}}$-many two-to-one multiplexers (MUX1) and is stored in ME$_s$ registers. Next, as we previously discussed, a customized multiplexer is assigned for each specific operand ($c^{\ell}_{\text{in.a}}$ or $c^{\ell}_{\text{in.b}}$) of a given NTT core (MUX2) to select one of the coefficients within ME$_s$. Finally, MUX4 selects one of these two results based on which stage type is getting processed. A similar set of MUXs (MUX6 and MUX7) are used to reorder the data back in the original locations before storing the result in the memory.

Once the final results are ready to be written in the memory (ME4 and ME5), they will be stored in data memory during two consecutive clock cycles. In the last stage, however, ME4 and ME5 are stored in *output memory* in order to keep the data available for the next module(s).

Memory Utilization and Word-Packing. Storing multiple polynomial coefficients (or twiddle factors) in multiple parallel memory units (M20K) can cause memory block under-utilization both depth-size and width-size. Let us consider a general scenario where β-many numbers are stored in parallel; whether it is polynomial coefficients or twiddle factors.

Depth-wise: Each M20K memory unit holds 512-many 40-bit wide words and at any cycle, one word can be read from or written into the memory. When fewer than 512 words are stored in the memory, the rest of the memory rows cannot be used to store a secondary polynomial since at any point in time we are reading/writing one word associated with the first polynomial. As long as $\frac{n}{\beta} \geq 512$, M20K is fully utilized. This inequality generally holds in our hardware architecture except when $n = 2^{12}$ (smallest polynomial size) and $nc_{\text{NTT}} = 16$ which makes M20K *half* utilized. However, this is not an issue since our design is not BRAM-constrained when $n = 2^{12}$.

Width-wise: As the polynomial-size (n) grows, our design becomes more and more BRAM-constrained to the extent that at $n = 2^{14}$, there is not enough BRAM on the chip; thus, we have to use DRAM as well (we will discuss this in more detail in Section 5). Therefore, it is essential that the polynomials are efficiently stored in memory. By storing each coefficient in a separate physical BRAM, we will only reach $\frac{54}{2\cdot 40} = 68\%$ utilization. In contrast, we pack multiple coefficients and store them in fewer M20K units as shown in Figure 3 reaching memory utilization of $\beta \cdot 54/(\lceil \beta \cdot 54/40 \rceil \cdot 40)$. For $\beta = 8$, BRAM utilization will reach more than 98%.

Performance. The NTT module processes one stage in $\frac{n}{2nc_{\text{NTT}}}$ cycles. Computing the NTT of a polynomial takes $\log n$ stages, and hence, it takes $\frac{n \log n}{2nc_{\text{NTT}}}$ cycles to compute one NTT.

INTT Module. Our NTT architecture can be used for both NTT and INTT with only a few modifications: (i) the NTT core is replaced by INTT core (see Figure 3), (ii) the control unit operates in the reverse order of stage numbers, and (iii) twiddle factor memories store the corresponding precomputed values. The rest of the architecture remains unchanged.

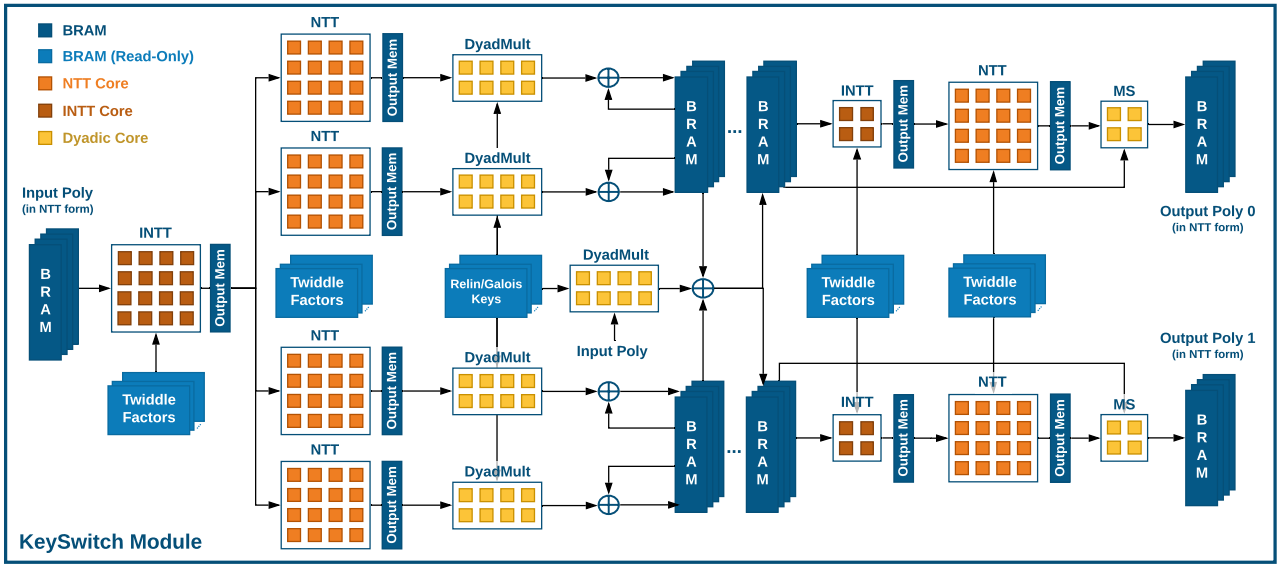

Figure 5: Architecture of KeySwitch module.

4.3. KeySwitch Module

`KeySwitch` is the most computationally intensive high-level operation in the CKKS scheme. It has several important roles, including relinearization and ciphertext rotation. In this section, we describe our proposed architecture for this operation as well as the challenges we faced during the design process.

Figure 5 illustrates the KeySwitch architecture. From the functionality perspective, this architecture corresponds to Algorithm 7. To reduce on-chip memory usage, the design takes one polynomial (one RNS component) at a time and outputs two polynomials. Recall that in CKKS, all polynomials are in NTT form by default. Thus, once the input polynomial is written into the input memory, it has to be converted back to the original domain. This process is performed using the first INTT module (INTT_0). Next, the polynomial is transformed to the NTT form for all other primes (including the special prime). Since per each INTT computation, we have to perform k NTT, the throughput of the NTT module(s) has to be k-times the throughput of INTT_0. Here, k is the number of RNS components of ciphertext modulus, i.e., $L+1$. This requirement can be realized in two different ways: (i) having one NTT module with k-many more cores than INTT_0 or (ii) having multiple NTT module with fewer cores per each module. We denote this NTT module (or a set of them) as NTT_0. We will discuss the trade-offs later in this section. In Figure 5, the second approach (using more than one NTT module) is chosen for $n = 2^{13}$ and $k = 4$ parameter set.

In the NTT/INTT module, the intermediate results are read/written from internal memory and the final results are written to the *output memory*. Once the NTT computations are finished, the DyadMult module computes the dyadic product between the output of NTT modules and the relinearization/Galois keys according to Algorithm 7. Recall that a dyadic product on the original input polynomial is also needed in `KeySwitch`; thus, a separate Dyadic module is used. After dyadic product computation, the result is stored in the corresponding memory banks. There are two *sets* of BRAM banks, each bank containing the RNS components of one polynomial.

The computation flow described above repeats for k-many times (one per each RNS component). The result is accumulated in the BRAM banks. After k iterations, the second part of the computation – usually referred to as *Modulus Switching* (developed in [12]) – is performed. In Modulus Switching which executes `Floor`, the polynomial corresponding to the *special prime* has to be transformed back to the time domain (by INTT_1) and then be transformed using every other k primes (by NTT_1). The aforementioned process is independently performed for both sets of banks. Next, the polynomial is multiplied by the inverse value of the associated prime and subtracted from the result of the first half of KeySwitch computation. The MS module embeds multiplication and subtraction operations. The output of KeySwitch is stored as two sets of k polynomials referred to as "Output Poly 0/1" in Figure 5.

Balancing Throughput. Our primary goal in designing KeySwitch architecture is to have a fully *end-to-end pipelined* module that can process many key switching operations simultaneously without any (or excessive) FIFOs between different components. Thus, we have to tune the throughput of each component carefully. As we discussed in Section 4.2, this is one of the reasons to design a flexible architecture for NTT, the throughput of which can be adjusted based on the number of cores that it encompasses. According to Algorithm 7, per each initial INTT, we have to compute k NTTs. In the next part, we will discuss two main approaches to realize this requirement.

Number of Cores vs. Number of Modules: To balance the throughput of INTT_0 and NTT_0, one can allocate a single NTT module with $nc_{\text{NTT}_0} = k \cdot nc_{\text{INTT}_0}$. However, in practice, having a very large NTT module with more than 32 cores

results in place-and-route failures during hardware synthesis. Moreover, recall from Section 4.2 that the ALM consumption of NTT module grows faster than linear with respect to the number of cores: $\mathcal{O}\left(nc_{\text{NTT}_0} \log nc_{\text{NTT}_0}\right)$. Therefore, breaking total number of NTT cores into separate modules results in less ALM consumption and more robust hardware synthesis. The downside of this approach is more BRAM utilization since each NTT module has its own internal data memory (for holding intermediate values) as well as the output memory.

Considering all other restrictions and requirements, we choose multiple NTT modules over a single big one. In general, let's denote the number of NTT_0 as m_0 (assuming a power of two number). Thus, we have: $nc_{\text{NTT}_0} = k \cdot nc_{\text{INTT}_0}/m_0$.

Next, we can compute the number of cores needed for DyadMult module. Recall from Section 4.2 that it takes $(n \log n)/(2nc_{\text{NTT}_0})$ cycles for NTT module to finish the computation. The DyadMult module has to compute the product of NTT output with two different sets of keys ($\texttt{ksk} = \mathbf{D}_0 \mid \mathbf{D}_1$). It takes $(2n)/nc_{\text{DYD}}$ cycles to perform dyadic multiplication on the output of the NTT module. Since in general, $\log n$ is not a power of two, the throughputs do not perfectly match. We make sure that the throughput of Dyadic module is greater than that of (or equal to) the NTT module by satisfying the following inequality:

$$\frac{2n}{nc_{\text{DYD}}} \leqslant \frac{n \log n}{2nc_{\text{NTT}_0}} \Rightarrow nc_{\text{DYD}} = \left\lceil \frac{4nc_{\text{NTT}_0}}{\log n} \right\rceil$$

The throughput of INTT_1 modules can be adjusted by assigning $nc_{\text{INTT}_1} = \lceil nc_{\text{INTT}_0}/k \rceil$. One can also determine $nc_{\text{NTT}_1} = nc_{\text{INTT}_0}$ and $nc_{\text{MS}} = \lceil (2nc_{\text{NTT}_1})/\log n \rceil$. For two FPGA chips that we have implemented HEAX on, the optimal architecture parameters are computed and summarized in Table 5.

Simultaneous KeySwitch Ops. and Synchronization. Figure 6 shows the high-level pipeline of KeySwitch module for $n = 2^{13}$ (third row of Table 5). All of the modules – and their internal components – are pipelined, and the throughput is balanced. As can be seen, multiple key switching operations are computed simultaneously in different pipeline stages (in lighter colors). The fifth Dyadic module that operates on input polynomial BRAM is *synchronized* with the rest of the Dyadic modules even though the computation can be started as soon as the input poly is available. The reason is that during each of the k iterations of Dyadic product, each module computes and accumulates the results by reading/writing from/to a separate BRAM bank. This enables us to avoid any *memory replication* considering that these memory banks are prohibitively large. However, this *delayed* computation introduces a dependency problem in the pipeline referred to as "Data Dependency 1". By the time the k-th Dyadic module starts the computation, the content of input poly is overridden by the next key switching operation. As a result, we allocate enough BRAM to hold f_1-many polynomials. The value of f_1 depends on the architecture parameter, more precisely: $f_1 = \left\lceil 3 + \frac{nc_{\text{INTT}_0}}{nc_{\text{NTT}_0}} \right\rceil$.

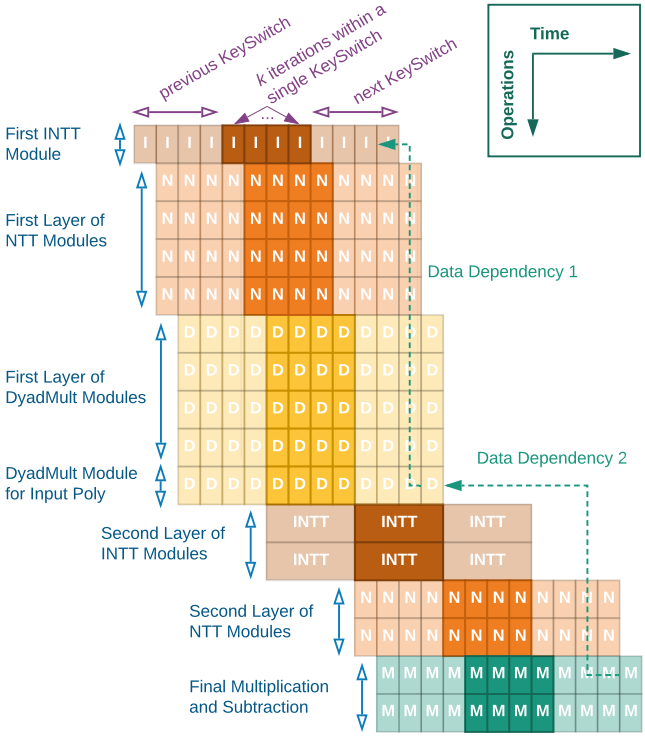

Figure 6: High-level pipeline of KeySwitch module.

Similarly, one of the inputs of the MS module is coming from the output of DyadMult modules, which will be overridden by subsequent results. This is marked as "Data Dependency 2" in Figure 6. Thus, we need to allocate more memory to store the outcome of the DyadMult modules in f_2 different buffers. The value of f_2 can be computed as

$$f_2 = \left\lceil 1 + m_0 \cdot \frac{nc_{\text{INTT}_1}}{nc_{\text{NTT}_1}} + \frac{nc_{\text{INTT}_1} \cdot \log n}{nc_{\text{MS}}} \right\rceil$$

Memory Read/Write Rate Conversion. Recall that we pack and store multiple words in parallel to minimize BRAM utilization while supporting simultaneous accesses. Given that each module is consuming and producing a different number of words at a time, we need to make any two subsequent modules compatible in terms of memory read/write rate.

Down-Scale Conversion: This procedure is needed when the number of words that have to be read in the next module is lower than the current module. For example, going from NTT_0 to DyadMult, we need to reduce the number of values we store in parallel. We satisfy this constraint by writing the output into multiple M20K units without *overlap*. This inevitably results in lower BRAM saturation both width-wise and depth-wise.

Up-Scale Conversion: This procedure is used, for example, between INTT_1 and NTT_1. Consider that the conversion rate is r, meaning that every r cycles, the result of previous block has to be written into *one* memory row. Therefore, during r-many cycles, the results are temporarily stored in registers and they will be written into certain number of M20Ks in parallel. We leverage the data compaction approach that we described in Section 4.2 in order to increase BRAM saturation.

5. SYSTEM-VIEW and DATA FLOW

In this section, we discuss a higher-level view of the computation and elaborate on the data flow. Figure 7 shows a system-view comprising host CPU and FPGA Board which are connected via Peripheral Component Interconnect express (PCIe) bus. On FPGA board, exist the FPGA chip as well as off-chip DRAM memory connected via DDR interface.

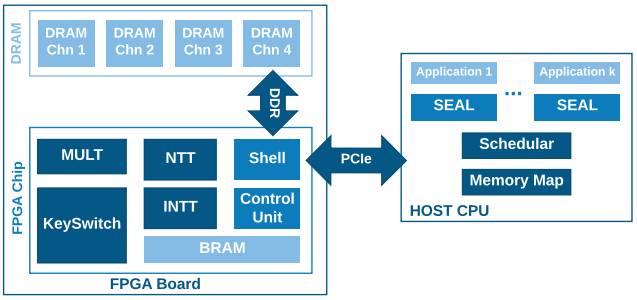

Figure 7: System-view.

5.1. On-Chip vs. Off-Chip Memory Accesses

There are two main ways to store data on FPGA board: (i) *off-chip* DRAM with several Gigabytes of capacity but high response delay and (ii) *on-chip* BRAM with few Megabits of capacity but very fast response time and high throughput. As has been shown by prior art [66, 67], leveraging off-chip memory to store intermediate results significantly reduces the overall performance due to high delays between subsequent reads and writes. One of our primary design goals is to avoid off-chip memory access as much as possible. We have introduced several techniques to use minimal on-chip memory, re-use many BRAM units, together with data compaction (see Section 4.2 and Section 4.3). As a result, no off-chip memory access is performed for $n = 2^{12}$ parameter set on both Arria 10 and Stratix 10 FPGAs; which is one of the main reasons for our unprecedented performance improvements.

For $n = 2^{13}$ parameter set, there is sufficient on-chip memory on Stratix 10 chip. Unfortunately, for $n = 2^{14}$, there is not enough BRAM available for our design, and as a result, we have to move some part of the data to off-chip memory. In order to minimize the effect of off-chip memory accesses, we choose to put key switching keys (`ksk`) on DRAM because of two main reasons: (i) the size of these keys grow very rapidly with HE parameters. In general, the size of `ksk` grows as $\mathcal{O}(nk^2)$, and roughly speaking, k grows linearly with n which results in (almost) $\mathcal{O}(n^3)$. This is the highest growth rate compared to all other memory components, including twiddle factors which grow as $\mathcal{O}(nk)$. (ii) `ksk` is only read once per each `KeySwitch`. Note that each unique element of twiddle factors is read k times during one KeySwitch operation; thus, twiddle factors are less suitable candidates.

We distribute the `ksk` among four different DRAM banks such that at any point in time, the full capacity of off-chip memory bandwidth is used. In order to further mask the effect of DRAM accesses, we leverage the *burst mode* in which a long sequence of data is read at the same time on each channel. The entire process of reading `ksk` from DRAM is pipelined to minimize the drop in throughput of KeySwitch. It is worth-mentioning that DRAM bandwidth is sufficient to match the throughput of KeySwitch. Per each `KeySwitch`, two sets of `ksk` have to be streamed to FPGA chip. Each of these sets, hold $k \cdot (k+1)$-many vectors of size n. Substituting $n = 2^{14}$, $k = 8$, and 64-bit per each word results in ≈ 151 Mega bits. We have to stream this volume of data in 383 microseconds (please see Table 8). Therefore, DRAM bandwidth should be higher than 49.28 GBps, which is indeed lower than the measured bandwidth of all four channels combined.

In addition to storing `ksk`, we use DRAM for one more purpose. In some applications, it is more efficient to store the result of computation on DRAM instead of sending them back to CPU in case these results are going to be used later on. The address at which the result is stored is held on the CPU side and is shown as "Memory Map". The memory map is used to point to the ciphertext(s) that are stored on DRAM to be used during the rest of the computation without involving PCIe.

5.2. Data Transfer on PCIe

In order to maximize the utilization of computation blocks on FPGA, we need to interleave computation and data transfer between FPGA and CPU. We divide this design process into two parts: CPU-side and FPGA-side. On the CPU-side, we need to sequence and batch multiple operations in the program (that uses SEAL) and start the data transfer process on PCIe using *multiple threads*. On the FPGA-side, we need to allocate the necessary buffers to store the data received from CPU.

Sequencing and Batching. Transferring data on PCIe involves three main steps: (i) a `memcpy` is issued to copy the content of the polynomial to pinned memory pages, (ii) CPU signals FPGA that the data is ready, and (iii) FPGA reads the data from PCIe. In order to reduce the time that takes to copy the data, Direct Memory Access (DMA) is used. However, even by relying on DMA, the maximum throughput that PCIe can provide depends on the message size and the number of simultaneous data transfer requests. Therefore, we transfer (at least) a complete polynomial ($2^{15} - 2^{17}$ Bytes) in each request. Moreover, we implement a multi-threaded data transfer mechanism that uses eight threads to interleave eight separate polynomials at a time to maximize the PCIe throughput and avoid unnecessary bubbles in the computation pipeline.

Double and Quadruple Buffering. For the MULT module, it suffices to double-buffer the input such that CPU writes to one of these buffers and FPGA reads from the other one. For KeySwitch module, however, we need to perform quadruple buffering due to the data dependency on input polynomial as discussed in Section 4.3. In order to make sure buffers are not overridden before they are read, we stop the writing process if the buffer has not been read yet.

6. IMPLEMENTATION and EXPERIMENTS

6.1. Experimental Setup

In this section, we elaborate in detail on the resource consumption of each component of our design starting from computation cores and moving to basic and high-level modules. We also provide our experimental results and compare the performance of HEAX to Microsoft SEAL execution on CPU.

We implement HEAX on two Intel FPGAs considering two different scales to illustrate the adaptability of HEAX:

• *Board-A:* with an Intel Arria 10 GX 1150 chip, 4 GB of DRAM memory with two independent DRAM channels, and PCIe Gen3 with eight lanes providing 7.88 GBps bandwidth in each direction from host CPU to FPGA and vice versa.

• *Board-B:* with Intel Stratix 10 GX 2800 chip, 64 GB of DRAM memory with four independent DRAM channels, and PCIe Gen3 with 16 lanes enabling 15.75 GBps bandwidth in each direction from host CPU to FPGA and vice versa. Each DRAM channel has a unidirectional bandwidth of 16 GBps based on the DDR4 interface.

There are three major types of resources that are available on the FPGA chip:

• Digital Signal Processing (DSP) units that are able to perform one 27-bit or two 18-bit multiplications.

• Adaptive Logic Modules (ALM) that represent core logic units and provide two combinational adaptive look-up tables, a two-bit full adder, and four Registers (REG) that are capable to hold a 1-bit value each.

• Block RAM (BRAM) units that represent on-chip memories for fast read and write operations. BRAM consists of M20K units that hold 512-many 40-bit values.

Table 1 summarizes the breakdown of resources available on each FPGA chip as well as the connections to DRAM.

Board	Chip	Chip Resources					DRAM	
		DSP	REG	ALM	BRAM		#chnl.	BW (GBps)
					bits	#M20K		
Board-A	Arria 10 GX 1150	1518	1.71M	427K	53Mb	2.7K	2	34
Board-B	Stratix 10 GX 2800	5760	3.73M	933K	229Mb	11.7K	4	64

Table 1: Summary of FPGA boards' specifications.

We evaluate our design on a wide range of HE parameters: from ciphertext polynomial size (n) of 2^{12} and 109-bit ciphertext modulus ($\lfloor \log qp \rfloor + 1$) to 2^{14} with 438-bit ciphertext modulus. We refer to these parameter sets as Set-A, Set-B, and Set-C, respectively (summarized in Table 2). Recall that k is the number of small RNS components of ciphertext modulus.

HE Param. Set	n	$\lfloor \log qp \rfloor + 1$	k
Set-A	2^{12}	109	2
Set-B	2^{13}	218	4
Set-C	2^{14}	438	8

Table 2: Description of HE parameter sets. n is the ciphertext polynomial size, qp is the ciphertext modulus, and k is the number of RNS components of q.

These parameters are selected based on the HE security standards [1] for 128-bit classical security. Parameters with 128-bit post-quantum security require slightly smaller ciphertext moduli, hence are easier to implement on an FPGA. We selected as few prime moduli for RNS as possible since they are critical to performance according to analysis in [37]. Note that parameter sets corresponding to 2^{11} (or lower) are almost never used in practice due to the multiplication depth of 1 (or zero). Choosing 2^{15} (or higher) results in enormous computation blow-up and are also rarely used in practice.

We compare the performance of HEAX with Microsoft SEAL V3.3 [69], which is an FHE library for BFV and CKKS schemes that has undergone several years of performance optimizations. We measure the performance of SEAL on a single-threaded Intel Xeon(R) Silver 4108 running at 1.80 GHz; which is a similar CPU used in prior art [67].

6.2. Resource Consumption

Computation Cores. Table 3 provides a detailed resource consumption of Dyadic, NTT, and INTT computation cores as well as the number of pipeline stages (delay) for each core.

Core Name	DSP	REG	ALM	#Stages
Dyadic	22	4526	1663	23
NTT	10	6297	2066	50
INTT	10	5449	2119	49

Table 3: Resource consumption of each core type.

Basic Modules. Table 4 provides a detailed resource consumption of shell as well as different modules for a various number of cores within the module. The BRAM utilization results are reported for Set-B parameters ($n = 2^{13}$). As can be seen, BRAM *bits* usage in each module does not depend on the number of cores. However, as the number of core grows, more coefficients are stored in parallel which results in consuming more M20K units. In the last column, the number of cycles that takes for each module to process a polynomial (or pair of polynomials in case of MULT module) is reported.

Module	#Cores	DSP	REG	ALM	BRAM		Cycles
					#bits	#M20K	
A10 Shell	-	1	79203	39222	886496	144	-
S10 Shell	-	2	86984	45612	1201096	173	-
MULT	4	88	42817	15795	1104384	65	1024
	8	176	61878	22160		65	512
	16	352	93594	35257		164	128
	32	704	181503	62157		293	64
NTT	4	40	61670	22316	1514496	86	6144
	8	80	96919	36336		185	3072
	16	160	196205	67865		380	1536
	32	320	387357	142300		725	768
INTT	4	40	63917	22700	1514496	86	6144
	8	80	104575	37331		185	3072
	16	160	182478	68645		380	1536
	32	320	384267	144957		724	768

Table 4: Resource consumption of basic modules.

Complete Design. Table 6 provides a breakdown of FPGA resource consumption for different HE parameter sets. The complete design encompasses the KeySwitch module along with the MULT module. For standalone NTT requests from CPU, the NTT modules within KeySwitch is used.

FPGA Device	HE Param. Set	KeySwitch Architecture Parameter Set
Arria10	$n = 2^{12}$ (Set-A)	$1 \times \text{INTT}^{(8)} \rightarrow 2 \times \text{NTT}^{(8)} \rightarrow 3 \times \text{Dyad}^{(4)} \rightarrow 2 \times \text{INTT}^{(4)} \rightarrow 2 \times \text{NTT}^{(8)} \rightarrow 2 \times \text{Mult}^{(2)}$
Stratix10	$n = 2^{12}$ (Set-A)	$1 \times \text{INTT}^{(16)} \rightarrow 2 \times \text{NTT}^{(16)} \rightarrow 3 \times \text{Dyad}^{(8)} \rightarrow 2 \times \text{INTT}^{(8)} \rightarrow 2 \times \text{NTT}^{(16)} \rightarrow 2 \times \text{Mult}^{(4)}$
	$n = 2^{13}$ (Set-B)	$1 \times \text{INTT}^{(16)} \rightarrow 4 \times \text{NTT}^{(16)} \rightarrow 5 \times \text{Dyad}^{(8)} \rightarrow 2 \times \text{INTT}^{(4)} \rightarrow 2 \times \text{NTT}^{(16)} \rightarrow 2 \times \text{Mult}^{(4)}$
	$n = 2^{14}$ (Set-C)	$1 \times \text{INTT}^{(8)} \rightarrow 4 \times \text{NTT}^{(16)} \rightarrow 5 \times \text{Dyad}^{(8)} \rightarrow 2 \times \text{INTT}^{(1)} \rightarrow 2 \times \text{NTT}^{(8)} \rightarrow 2 \times \text{Mult}^{(4)}$

Table 5: KeySwitch architecture for different HE parameter sets.

FPGA Device	HE Param. Set	DSP (%)	REG (%)	ALM (%)	BRAM bits (%)	BRAM #M20K (%)	Freq. (MHz)
Arria10	Set-A	1185 (78)	723188 (42)	246323 (58)	26596320 (48)	1731 (64)	275
Stratix10	Set-A	2018 (35)	1554005 (42)	582148 (62)	26907592 (11)	3986 (34)	300
	Set-B	2610 (45)	1976162 (53)	698884 (75)	201332624 (84)	10340 (88)	300
	Set-C	2370 (41)	1746384 (47)	599715 (64)	182847524 (76)	9329 (80)	300

Table 6: Resource consumption of HEAX for different HE parameter sets.

FPGA Device	HE Param. Set	NTT			INTT			Dyadic MULT		
		CPU	HEAX	Speed-up	CPU	HEAX	Speed-up	CPU	HEAX	Speed-up
Arria10	Set-A	7222	89518	12.4	7568	89518	11.8	36931	1074219	29.1
Stratix10	Set-A	7222	195313	27.0	7568	195313	25.8	36931	1171875	31.7
	Set-B	3437	90144	26.2	3539	90144	25.5	18362	585938	31.9
	Set-C	1631	41853	25.7	1659	41853	25.2	9117	292969	32.1

Table 7: Performance comparison of HEAX with CPU. Number of operations per second for CKKS *low-level* operations.

FPGA Device	HE Param. Set	KeySwitch			MULT+ReLin		
		CPU	HEAX	Speed-up	CPU	HEAX	Speed-up
Arria10	Set-A	488	44759	91.7	420	44759	106.6
Stratix10	Set-A	488	97656	**200.5**	420	97656	**232.5**
	Set-B	97	22536	**232.3**	84	22536	**268.3**
	Set-C	16	2616	**163.5**	15	2616	**174.4**

Table 8: Performance comparison of HEAX with CPU. Number of operations per second for CKKS *high-level* operations.

6.3. Performance

Critical Paths and Maximum Clock Frequency. The FPGA operating clock frequency directly affects the performance; thus, we have analyzed the critical paths of our design and have eliminated such paths during many design iterations. These modifications include but are not limited to: altering the data flow of computation and reducing the fan-out as well as data dependency among registers, breaking large multiplexers into smaller ones, and minimizing the number of levels of logic per pipeline stage. The final maximum clock frequency that are achieved for Arria 10 and Stratix 10 FPGA chips are 275 MHz and 300 MHz, respectively.

Scalability. One of the desirable features of HEAX is that it can automatically be instantiated at different scales (with no manual tuning) based on the available hardware resources. To illustrate the scalability and adaptability of our design, we have instantiated HEAX for the same HE parameters (Set-A) but at two different scales (see Table 5). As can be seen from Table 6, the up-scaled instantiation on Stratix 10 consumes (close to) twice the resources and provides twice the throughput compared to Arria 10 instantiation (see Table 8). This property enables cloud providers to seamlessly instantiate HEAX based on the underlying application, HE parameters, and available FPGA resources.

Performance Comparison. Table 7 shows the performance results (number of operations per second) of HEAX for low-level operations and its comparison with Microsoft SEAL execution on CPU. The performance results are reported for processing a single polynomial (in case of NTT/INTT) or pair of polynomials (Dyadic MULT). On Stratix 10, 16-core modules are instantiated for NTT/INTT and MULT. On Arria 10, a 16-core MULT and 8-core NTT/INTT modules are used (see Table 5). Note that we report the performance results for low-level operation merely for completeness. These operations are rarely used in isolation and are instead used as part of high-level operations. For high-level operations, i.e., Rotation and Relinearization (using KeySwitch) and a complete ciphertext multiplication (using MULT and KeySwitch), the performance improvements are more pronounced; because many of such operations can be executed in parallel compared to CPU.

Table 8 shows the performance results as well as the speed-up for high-level operations in CKKS scheme. As can be seen, HEAX achieves close to *two orders of magnitude* performance improvement using Arria 10 FPGA (Board-A) compared to CPU (first row of Table 8). On a more powerful FPGA, i.e., Intel Stratix 10 (Board-B), HEAX achieves **164-268**$\times$ performance improvements among various HE parameter sets compared to CPU (second to fourth rows of Table 8).

7. RELATED WORK

Homomorphic encryption can fundamentally change the trust and computation model of many applications and industries. For example, in Machine Learning as a Service (MLaaS), the privacy of users' data as well we the confidentiality of the cloud server's ML model can be preserved. Privacy-preserving MLaaS has been studied rigorously in recent years and several protocol-level and algorithmic improvements have been proposed [36, 48, 63, 46, 19, 64, 62, 61, 42]. However, the computation overhead compared to the traditional plaintext execution remains a standing challenge.

The CKKS scheme is one of the most recently proposed FHE schemes that allows homomorphic operations on fixed-point numbers; making it the prime candidate for machine learning applications. To the best of our knowledge, no hardware architecture has been proposed for the CKKS scheme, and in this paper, we propose the first of its kind. As a result, it is not fair to compare the performance of HEAX with previous designs that focus on non-CKKS schemes. In what follows, we briefly review the research effort related to FPGA, ASIC, and GPU-based acceleration for non-CKKS schemes as well as (non-HE) secure computation protocols.

Hardware Acceleration for non-CKKS Schemes. In [66], a system based on FPGA is proposed for BFV scheme to process ciphertext polynomial sizes of 2^{15}. However, due to the massive off-chip data transfer, their design does not yield superior performance compared to CPU execution.

Perhaps, the closest work to ours is by Roy et al. [67] in which authors propose an architecture for BFV scheme and implement their design on Xilinx Zynq UltraScale+ MPSoC ZCU102. In order to avoid off-chip memory accesses, authors focus on $n = 2^{12}$ ciphertext sizes and report $13\times$ speed-up (using two instances of their proposed processors) compared to the FV-NFLlib [33] executing on an Intel i5 processor running at 1.8 GHz. However, compared to a more optimized Microsoft SEAL library [68], FV-NFLlib is $1.2\times$ slower [6]. In addition, our design is significantly more modular and scalable. We have instantiated HEAX for three different set of HE parameters with no manual tuning (polynomial sizes of 2^{12}, 2^{13}, and 2^{14}). Moreover, HEAX has a multi-layer pipelined design and is drastically more efficient, offering more than two orders of magnitude performance improvement compared to Microsoft SEAL running on Intel Xeon Silver 4108 at 1.8 GHz (note that similar processor is used compared with [67] running at identical frequency).

FPGA-based Co-Processors. Designing co-processors has also been studied in the literature. These co-processors work in conjunction with CPUs and accelerate one or more of the homomorphic operations [45, 55, 21, 38, 56, 49]. In [55] and [38, 56], authors focus on designing hardware architecture for the *encryption* operation only, by leveraging Karatsuba and Comba multiplication algorithms, respectively. In [21], a Homomorphic Encryption Processing Unit (HEPU) is proposed for LTV scheme [53]. Authors focus on accelerating the Chinese Remainder Transform (CRT) for power-of-2 cyclotomic rings and report 3.2-4.4$\times$ performance improvements for homomorphic multiplication using Xilinx Virtex-7 FPGA.

Large-Integer Multiplication Hardware Acceleration. A line of research focuses on designing very large integer multipliers – based on FPGAs or ASICs – that can be used to accelerate homomorphic operations [74, 75, 28, 29, 13]. These architectures support 768K-bit to 1.18M-bit multiplications.

In [14], a large-integer multiplier and a Barrett modular reduction are proposed that can accelerate HE operations by $11\times$. However, the performance degradation due to the off-chip memory accesses are not considered in this work which can be the main bottleneck as reported by prior art [67, 60]. In contrast, HEAX does not rely on large-integer multiplication, and instead, leverages the parallelism based on RNS to achieve efficient routing and superior performance.

GPU-based HE Acceleration. Graphics Processing Unit (GPU) is an alternative computing platform to accelerate evaluation in HE [25, 22, 57, 5, 50, 72]. Wang et al. [72] have proposed the first GPU acceleration of FHE that targets Gentry-Halevi [35] scheme. Subsequent improvements are reported in [73]. In [71], a GPU-based implementation of BGV scheme [11] is introduced. In [5], a comprehensive study is reported for multithreaded CPU execution as well as GPU implementation of the BFV scheme. To the best of our knowledge, there is no GPU-accelerated implementation of the CKKS scheme. GPUs normally offer less performance per watt of power than FPGAs by design. Therefore, FPGAs are more suitable candidates for high-performance, low-power secure computation in the cloud.

Acceleration of YASHE and LTV Schemes. Several works [58, 24, 27, 59, 20, 21] focus on improving the performance of YASHE [9] and LTV [53] schemes or their variants. These constructions – based on an overstretched NTRU assumption – are subject to a subfield lattice attack [2] and are no longer secure. Nevertheless, the contribution in optimization techniques remain valuable.

In what follows, we briefly review two of the most relevant designs to ours. In [65], an architecture for YASHE scheme is proposed that provides $25\times$ performance improvement over CPU. However, authors assume unlimited memory bandwidth which renders off-chip memory accesses free of cost and is not a realistic assumption. Pöppelmann et al. [60] have also proposed an architecture for YASHE scheme. Since ciphertexts are prohibitively large to be stored on on-chip memory, authors propose to leverage the idea of Cached-NTT [3, 4]. Cached-NTT suggests formulating the NTT of large polynomials as many invocations of a *fixed* computation with the property that loading/saving intermediate results from/to off-chip memory is minimized, thus, the notion of *cached*. Cached-NTT mini-

mizes the number of times off-chip memory transactions are performed since the off-chip memory access is significantly costlier compared with on-chip memory access. In contrast, in HEAX, we rely on the ring isomorphism property and perform independent computation on each RNS component of large ciphertext. This, in turn, allows us to avoid off-chip memory accesses for small-size HE parameters and minimize such accesses for large parameters.

Hardware Acceleration of Secure Computation Protocols. Secure Multi-Party Computation (SMPC) protocols can also be used for the task of privacy-preserving MLaaS. The Garbled Circuits protocol [76] is one of the secure *two-party* computation protocols for which FPGA-based accelerators have been proposed [43, 32, 44]. However, compared to HE, the communication between parties is significantly higher, making the end-to-end protocol mostly bandwidth constrained. Thus, accelerating the *computation* portion of SMPC protocols does not necessarily reduce the execution time of the protocol in general. In contrast, the main bottleneck of HE is the computation overhead and reducing the homomorphic evaluation time directly increases the practicality of HE-based computations.

8. CONCLUSION

In this paper, we introduced a novel set of architectures for Fully Homomorphic Encryption (FHE). To the best of our knowledge, HEAX is the *first* architecture and fully-implemented hardware acceleration for the CKKS FHE scheme. CKKS is the prime candidate for machine learning on encrypted data due to floating-point support of this scheme. The components designed in HEAX can also be used for other lattice-based cryptosystems and other FHE/HE schemes. The proposed architecture provides a unique degree of flexibility that can be readily adjusted for various FPGA chips. As a proof-of-concept, we have implemented HEAX on two different FPGAs with contrasting hardware resources. Moreover, unlike prior FPGA-based acceleration for BFV scheme, our design is not tied to a specific FHE parameter set. We evaluate HEAX on a wide range of FHE parameters demonstrating more than *two orders of magnitude* performance improvements. We hope that HEAX paves the way for large-scale deployment of privacy-preserving computation in clouds.

References

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[2] Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 153–178, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[3] Bevan M Baas. *An approach to low-power, high-performance, fast Fourier transform processor design*. PhD thesis, Citeseer, 1999.

[4] Bevan M Baas. A generalized cached-FFT algorithm. In *Proceedings.(ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, volume 5, pages v–89. IEEE, 2005.

[5] 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. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2019.

[6] 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, 2018. https://tches.iacr.org/index.php/TCHES/article/view/875.

[7] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 423–442, St. John's, NL, Canada, August 10–12, 2016. Springer, Heidelberg, Germany.

[8] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.

[9] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *14th IMA International Conference on Cryptography and Coding*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64, Oxford, UK, December 17–19, 2013. Springer, Heidelberg, Germany.

[10] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013: 16th International Conference on Theory and Practice of Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 1–13, Nara, Japan, February 26 – March 1, 2013. Springer, Heidelberg, Germany.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

[12] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, Palm Springs, CA, USA, October 22–25, 2011. IEEE Computer Society Press.

[13] Xiaolin Cao, Ciara Moore, Máire O'Neill, Elizabeth O'Sullivan, and Neil Hanley. Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction. *IACR Cryptology ePrint Archive*, 2013:616, 2013.

[14] Xiaolin Cao, Ciara Moore, Máire O'Neill, Neil Hanley, and Elizabeth O'Sullivan. High-speed fully homomorphic encryption over the integers. In *International Conference on Financial Cryptography and Data Security*, pages 169–180. Springer, 2014.

[15] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multikey homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. Cryptology ePrint Archive, Report 2019/524, 2019. https://eprint.iacr.org/2019/524.

[16] 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, *SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368, Calgary, AB, Canada, August 15–17, 2019. Springer, Heidelberg, Germany.

[17] 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, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031

of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.

[19] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster CryptoNets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

[20] David Bruce Cousins, John Golusky, Kurt Rohloff, and Daniel Sumorok. An FPGA co-processor implementation of homomorphic encryption. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[21] David Bruce Cousins, Kurt Rohloff, and Daniel Sumorok. Designing an FPGA-accelerated homomorphic encryption co-processor. *IEEE Transactions on Emerging Topics in Computing*, 5(2):193–206, 2016.

[22] cuFHE. `https://github.com/vernamlab/cuFHE`. Vernam Group.

[23] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[24] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating NTRU based homomorphic encryption using GPUs. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[25] Wei Dai and Berk Sunar. cuHE: A homomorphic encryption accelerator library. In Enes Pasalic and Lars R. Knudsen, editors, *Cryptography and Information Security in the Balkans*, pages 169–186, Cham, 2016. Springer International Publishing.

[26] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[27] Yarkın Doröz, Erdinç Öztürk, Erkay Savaş, and Berk Sunar. Accelerating LTV based homomorphic encryption in reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 185–204. Springer, 2015.

[28] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. Evaluating the hardware performance of a million-bit multiplier. In *2013 Euromicro Conference on Digital System Design*, pages 955–962. IEEE, 2013.

[29] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Transactions on Computers*, 64(6):1509–1521, 2014.

[30] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. A million-bit multiplier architecture for fully homomorphic encryption. *Microprocessors and Microsystems*, 38(8):766–775, 2014.

[31] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `http://eprint.iacr.org/2012/144`.

[32] Xin Fang, Stratis Ioannidis, and Miriam Leeser. Secure function evaluation using an FPGA overlay architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 257–266. ACM, 2017.

[33] FV-NFLlib. `https://github.com/CryptoExperts/FV-NFLlib`. CryptoExperts.

[34] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[35] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 129–148. Springer, 2011.

[36] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[37] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 83–105, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

[38] Shakirah Hashim and Mohammed Benaissa. Accelerating integer based fully homomorphic encryption using frequency domain multiplication. In *International Conference on Information and Communications Security*, pages 161–176. Springer, 2018.

[39] HEAAN. `https://github.com/snucrypto/HEAAN`. CryptoLab Inc.

[40] Jay Heiser and Mark Nicolett. Assessing the security risks of cloud computing. *Gartner report*, 27:29–52, 2008.

[41] HElib. `https://github.com/homenc/HElib`. IBM Corp.

[42] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. CryptoDL: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[43] Siam U Hussain and Farinaz Koushanfar. FASE: FPGA acceleration of secure function evaluation. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 280–288. IEEE, 2019.

[44] Siam U Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (MAC) on cloud servers. In *Proceedings of the 55th Annual Design Automation Conference*, page 33. ACM, 2018.

[45] Cedric Jayet-Griffon, M-A Cornelie, Paolo Maistri, PH Elbaz-Vincent, and Régis Leveugle. Polynomial multipliers for fully homomorphic encryption on FPGA. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[46] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1209–1222. ACM, 2018.

[47] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1209–1222, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[48] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[49] Alhassan KHEDR and Glenn Gulak. Homomorphic processing unit (HPU) for accelerating secure computations under homomorphic encryption, May 21 2019. US Patent App. 10/298,385.

[50] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. SHIELD: Scalable homomorphic implementation of encrypted data-classifiers. *IEEE Transactions on Computers*, 65(9):2848–2858, Sep. 2016.

[51] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11(4):83, Oct 2018.

[52] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16: 15th International Conference on Cryptology and Network Security*, volume 10052 of *Lecture Notes in Computer Science*, pages 124–139, Milan, Italy, November 14–16, 2016. Springer, Heidelberg, Germany.

[53] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *44th Annual ACM Symposium on Theory of Computing*, pages 1219–1234, New York, NY, USA, May 19–22, 2012. ACM Press.

[54] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[55] Vincent Migliore, Cédric Seguin, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat, and Russell Tessier. A high-speed accelerator for homomorphic encryption using the karatsuba algorithm. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):138, 2017.

[56] Ciara Moore, Máire O'Neill, Neil Hanley, and Elizabeth O'Sullivan. Accelerating integer-based fully homomorphic encryption using Comba multiplication. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014.

[57] nuFHE. `https://github.com/nucypher/nufhe`, NuCypher.

[58] Erdinç Öztürk, Yarkın Doröz, Erkay Savaş, and Berk Sunar. A custom accelerator for homomorphic encryption applications. *IEEE Transactions on Computers*, 66(1):3–16, 2016.

[59] Erdinç Öztürk, Yarkin Doröz, Berk Sunar, and Erkay Savas. Accelerating somewhat homomorphic evaluation using FPGAs. *IACR Cryptology ePrint Archive*, 2015:294, 2015.

[60] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 143–163. Springer, 2015.

[61] M. Sadegh Riazi and Farinaz Koushanfar. Privacy-preserving deep learning and inference. In *Proceedings of the International Conference on Computer-Aided Design*, page 18. ACM, 2018.

[62] M. Sadegh Riazi, Bita Darvish Rouhani, and Farinaz Koushanfar. Deep learning on private data. *IEEE Security and Privacy (S&P) Magazine*, 2019.

[63] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. *USENIX Security*, 2019, 2019.

[64] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707–721. ACM, 2018.

[65] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular hardware architecture for somewhat homomorphic function evaluation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 164–184. Springer, 2015.

[66] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Transactions on Computers*, 67(11):1637–1650, 2018.

[67] 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. IEEE, 2019.

[68] Microsoft SEAL (release 2.3). `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA.

[69] Microsoft SEAL (release 3.3). `https://github.com/Microsoft/SEAL`, June 2019. Microsoft Research, Redmond, WA.

[70] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[71] Wei Wang, Zhilu Chen, and Xinming Huang. Accelerating leveled fully homomorphic encryption using GPU. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2800–2803. IEEE, 2014.

[72] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using GPU. In *2012 IEEE conference on high performance extreme computing*, pages 1–5. IEEE, 2012.

[73] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers*, 64(3):698–706, 2013.

[74] Wei Wang and Xinming Huang. FPGA implementation of a large-number multiplier for fully homomorphic encryption. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pages 2589–2592. IEEE, 2013.

[75] Wei Wang, Xinming Huang, Niall Emmart, and Charles Weems. VLSI design of a large-number multiplier for fully homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):1879–1887, 2013.

[76] A. Yao. How to generate and exchange secrets. In *FOCS*. IEEE, 1986.