IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 451–480. DOI:10.46586/tches.v2024.i2.451-480

# CASA: A Compact and Scalable Accelerator for Approximate Homomorphic Encryption

Pengzhou He[1], Samira Carolina Oliva Madrigal[2], Çetin Kaya Koç[3,4,5], Tianyou Bao[1], and Jiafeng Xie[1]

[1] Department of Electrical and Computer Engineering, Villanova University, Villanova PA, USA
{phe,tbao,jiafeng.xie}@villanova.edu
[2] San José State University, San José, USA scolivamadrigal@gmail.com
[3] University of California Santa Barbara, Santa Barbara, USA cetinkoc@ucsb.edu
[4] Iğdır University, Iğdır, Turkey
[5] Nanjing University of Aeronautics and Astronautics, Nanjing, Republic of China

**Abstract.**
Approximate arithmetic-based homomorphic encryption (HE) scheme CKKS [CKKS17] is arguably the most suitable one for real-world data-privacy applications due to its wider computation range than other HE schemes such as BGV [BGV14], FV and BFV [Bra12, FV12]. However, the most crucial homomorphic operation of CKKS called key-switching induces a great amount of computational burden in actual deployment situations, and creates scalability challenges for hardware acceleration. In this paper, we present a novel **C**ompact **A**nd **S**calable **A**ccelerator (CASA) for CKKS on the field-programmable gate array (FPGA) platform. The proposed CASA addresses the aforementioned computational and scalability challenges in homomorphic operations, including key-exchange, homomorphic multiplication, homomorphic addition, and rescaling.
On the architecture layer, we propose a new design methodology for efficient acceleration of CKKS. We design this novel hardware architecture by carefully studying the homomorphic operation patterns and data dependency amongst the primitive oracles. The homomorphic operations are efficiently mapped into an accelerator with simple control and smooth operation, which brings benefits for scalable implementation and enhanced pipeline and parallel processing (even with the potential for further improvement).
On the component layer, we carry out a detailed and extensive study and present novel micro-architectures for primitive function modules, including memory bank, number theoretic transform (NTT) module, modulus switching bank, and dyadic multiplication and accumulation.
On the arithmetic layer, we develop a new partially reduction-free modular arithmetic technique to eliminate part of the reduction cost over different prime moduli within the moduli chain of the Residue Number System (RNS). The proposed structure can support arbitrary numbers of security primes of CKKS during key exchange, which offers better security options for adopting the scalable design methodology.
As a proof-of-concept, we implement CASA on the FPGA platform and compare it with state-of-the-art designs. The implementation results showcase the superior performance of the proposed CASA in many aspects such as compact area, scalable architecture, and overall better area-time complexities.
In particular, we successfully implement CASA on a mainstream resource-constrained Artix-7 FPGA. To the authors' best knowledge, this is the first compact CKKS accelerator implemented on an Artix-7 device, e.g., CASA achieves a 10.8x speedup compared with the state-of-the-art CPU implementations (with power consumption of only 5.8%). Considering the power-delay product metric, CASA also achieves 138x and 105x improvement compared with the recent GPU implementation.


Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

**Keywords:** Approximate homomorphic encryption CKKS · compact and scalable · FPGA · hardware accelerator · NTT · memory · partially reduction-free · RNS

# 1 Introduction

Data-privacy related technology advancement has drawn noted attention from various communities recently. Fully Homomorphic Encryption (FHE) represents one of the promising data-privacy technologies as it can execute various compuational functions over encrypted data [Gen09a]. Subsequently, FHE is becoming significantly popular since its original introduction in [Gen09b]. Thus, more variants of FHE schemes and related implementation works have been proposed in recent years [MKS+22, RLPD20, HZL+22].

HEAAN or equally referred to as CKKS [CKKS17] is the first FHE scheme that supports real number homomorphic arithmetic operations. The authors of CKKS also introduced a variant of CKKS based on the Residue Number System (RNS) to reduce the computational cost induced by large coefficients in CKKS [CHK+18]. This RNS variant reduced the coefficients' size from several hundreds of bits to a set of smaller coefficients within 64 bits, at the cost of increased complexity in homomorphic operations and new constraints of parameter selections. Meanwhile, there also exist other types of FHE schemes such as FV [FV12], BGV [BGV14], and BFV [Bra12]. Among these FHE schemes, RNS-CKKS (referred to as CKKS throughout the paper afterward) has obtained significant attention recently due to its fast operations on the encrypted real data [MKS+22].

## 1.1 Related Works, Motivation, and Proposed Plan

CKKS has been broadly implemented by software libraries across Windows, Linux, and Mac OS [ABBB+22]. However, the sheer amount of computational work makes it too expensive to use CKKS in today's deep-learning or even communication scenarios. In addition to limited reports for CKKS hardware accelerators in the literature, a compact & power-efficient cryptographic accelerator with decent processing performance is needed for practical scenarios that require low-degree FHE processing, including intelligent traffic, multi-party computation, and multi-party signature. The study of such accelerators in the literature is even more constrained. We put details about these applications in Section 5 (Paragraph of Application Discussion) for further reference.

The existing hardware accelerators for CKKS were either implemented through the application-specific integrated circuits (ASIC) or on the field-programmable gate array (FPGA) platform. While it is noticed that F1 [SFK+21], CraterLake [SFK+22], BTS [KKK+22], and ARK [KLK+22] are the four recent ASIC-implemented CKKS accelerators, the FPGA-based accelerators include some recent works like HEAX [RLPD20], coxHE [HZL+22], and Medha [MKS+22].

Nevertheless, the ASIC-based accelerators still suffer from large power and area usage. It is estimated by [MKS+22] that F1 [SFK+21] also contains unimplementable structures on practical FPGAs. CraterLake [SFK+22] also suffers from similar architectural drawbacks due to large resource consumption. As a result, F1 [SFK+21] and CraterLake [SFK+22] presented only simulation-based estimation results (similar to BTS [KKK+22] and ARK [KLK+22], based on the analysis in [MKS+22]). While the FPGA-based accelerators have demonstrated high throughput and low-latency implementations, they again involve large resource usage (let alone for fitting resource-constrained devices).

Overall, we observe that the existing hardware accelerators for CKKS involve three major drawbacks: (i) sophisticated structural design (which causes large resource usage); (ii) limited flexibility in scalability (difficult to adjust the accelerator for different application scenarios, especially for resource-constrained applications); (iii) lack of significant arithmetic innovation (key components are mostly implemented with existing algorithms/techniques).

Based on the above-mentioned limitations, in this work, we want to study a contrary accelerator design methodology, i.e., pursue simplicity and better performance per unit of cost. We found that sophisticated structures potentially increase synthesizing, routing, and placing challenges for modern hardware description language (HDL) compilers. Fine-tuned micro-architectures, on the other hand, provide better timing, area, and power performance. Hence, our aim was to create a hardware framework that utilizes finely tuned micro-architectures and facilitates parallel computation in a straightforward manner. This target, overall, requires three layers of effort. (i) First, we need to design optimized bottom-level units for primitive function units. To the best of authors' knowledge, we present the first fine-tuned micro-architecture for partially reduction-free arithmetic, which allows flexible moduli configuration during modulo operation. This component suits well the need for dealing with different moduli in the moduli chain of CKKS. (ii) Second, we need to design scalable modules for mid-level operations such as number theoretic transform (NTT), dyadic multiplication, and modulus switching. These scalable modules enable us to construct a smooth pipeline and highly occupied hardware for high-performance computation. (iii) Lastly, we need to create a balanced parallel computation strategy so that the processing capacities of individual components, including memory I/O bandwidth, can work well with each other. We categorize these steps into three layers: the arithmetic layer, which we describe in Section 3.1; the component layer, for which we describe individual modules in Section 3.2∼3.5; the architecture layer, for which we describe the accelerator from top-level and the proposed parallel computation strategy (Section 4).

## 1.2 Major Contributions

Following the proposed research plan, we design a generic architecture for CKKS accelerator based on a new design methodology with the help of several innovative techniques. The proposed accelerator is compatible with different CKKS parameter sets and suitable for different application scenarios. Major contributions of this work include:

- Unlike the existing design strategies, we present a new methodology for CKKS hardware design that emphasizes constructing fine-tuned micro-architecture and pertains to simplicity in each function module. We demonstrate this design methodology through detailed presentation of our design considerations and architectural schemes. The implemented accelerator overall obtains compact resource usage (while maintaining high-speed processing) on the FPGA devices.

- We showcase a low-latency parallel computation strategy for the bottleneck operation (key-switching) to facilitate the practical use value of CKKS. We describe an efficient pipeline data flow constructed by operating the area-optimized micro-architecture modules. This pipeline strategy demonstrates not only high efficiency but also desirable scalability (thanks to the scalable nature of the primitive modules). In this case, the entire accelerator has a broad implementation range, covering low- to high-ended hardware devices.

- We investigate the fundamental arithmetic in homomorphic operations and propose a novel micro-architecture for CKKS using the proposed modular partially reduction-free strategy. Compared with existing reduction techniques, we show that the proposed method reduces hardware computational complexity and can be considered the best fit for the moduli-chain requirement in CKKS.

- The proposed architecture and pipeline strategy allow many special prime slots for switching the keys. As the precision boundary for key-switching operation significantly depends on the size of special prime $P$, users of the proposed accelerator can adjust the precision threshold for key-switching to fit different practical requirements (which was not offered by the existing FPGA accelerators such as [RLPD20, HZL+22, MKS+22]).

- As a proof-of-concept, we implemented and evaluated the proposed design on FPGA devices. We demonstrate the superior performance of the proposed accelerator compared to the existing hardware accelerators. Even on the resource-constrained FPGA, we show a 26.4x speedup against the state-of-the-art CPU acceleration and a 161.6x power-delay improvement against a recent GPU acceleration for executing homomorphic multiplication.

Besides the above-mentioned contributions, we want to mention that the proposed CASA is the first hardware CKKS accelerator being implemented on the Artix-7 device with compact resource usage (decent timing) and scalable architecture options. This innovative achievement undoubtedly opens the door to accelerate the FHE schemes on resource-constrained lightweight applications. We hope the outcome of this work can also facilitate efficient acceleration of FHE schemes for practical lightweight applications such as the Internet-of-Things (IoT).

The rest of the paper is organized as follows. Preliminaries are described in Section 2. Primitive modules are presented in Section 3 and CASA is introduced in Section 4. Implementation & comparison are provided in Section 5. Conclusion is given in Section 6.

# 2 Preliminaries

This section provides a brief overview on the core algorithms and schemes used in this paper, covering from RNS to the HEAAN-variant on which our work is based.

## 2.1 Notations

Let $n$ be a power of two and $n = \deg(R)$ for integer polynomial ring $R = R[X]/(x^n + 1)$. $R_Q$ is further defined as $R \bmod Q$ for a positive integer $Q$. In this way, every coefficient of $R_Q$ is a number in the interval of $[0, Q-1]$. $Q$ is called the modulus of $R$. In the following sections, except otherwise stated, we use small letters, for example $a$, to denote polynomials in $R$. $\hat{a}$ is used to denote $a$ in the NTT domain, i.e., $\hat{a} = \mathrm{NTT}(a)$. Let $a[k]$ represent the $k$-th coefficient of $a$ for $k \in [0, n-1]$, while $a_i$ denotes the $i$-th RNS component of $a$ for $i \in [0, L]$. Finally, $L$ is the number of ciphertext levels in CKKS.

## 2.2 Residue Number System (RNS)

HE constructions built from cyclotomic rings have the inherent problem of dealing with modular arithmetic in rings of large characteristic. RNS is usually applied to help work with large coefficients in such schemes. Instead of computing modulo a large prime, we work in small rings and compute modulo small primes. One noted optimization is the double-CRT (Chinese remainder theorem) representation, a two layer representation. The double-CRT applies RNS representation to map an input cyclotomic polynomial into a vector of small polynomials with corresponding small moduli. Then each polynomial is mapped into the NTT domain as a vector modulo integers.

## 2.3 CKKS

CKKS embraces approximate arithmetic to maintain security of data. Compared with homomorphic schemes such as BGV [BGV14], FV [FV12], and TFHE [CGGI16], CKKS treats errors generated by approximate arithmetic as additional noise. This idea enables CKKS to endure many adversarial facts by considering them all as noises. Since it is not possible for homomorphic encryption schemes to achieve the IND-CCA1 (indistinguishability under (non-adaptive) chosen ciphertext attack) or IND-CCA2 (indistinguishability under adaptive chosen ciphertext attack) standard, the IND-CPA (indistinguishability

under chosen plaintext attack) standard is broadly chosen to be the security requirement for most homomorphic designs. CKKS relies on Ring Learning-with-Errors (RLWE) as the security base, which has been extensively studied and shown to be quantum resistant.

CKKS follows RLWE construction to choose $x^{n+1}$ as modulus polynomial for plaintext and ciphertext space. Plaintext $m \in R$ is an encoding of the message $z \in \mathbb{C}^{n/2}$. Notice that CKKS works with approximate arithmetic, so a scalar $\Delta$ (usually a large integer) is multiplied with each element of $z$ to maintain precision before encoding it into an element of $R$, i.e., the encoded information stored in $m$ is actually $\Delta z_i$. The scale operation will make a difference when users need to design their own leveled homomorphic circuits. For example, suppose ciphertexts $ct$ and $ct'$ encrypt $\Delta z$ and $\Delta z'$, respectively. When $ct$ and $ct'$ are homomorphically multiplied with each other, the correct homomorphic multiplication oracle will give out a ciphertext $ct''$ encrypting $\Delta^2 zz'$ rather than $\Delta zz'$. The original CKKS also provides a *rescaling*$(\cdot)$ oracle to divide the encrypted information by $\Delta$ [CHK+18].

Moreover, the lattice structure constructed by modular polynomial $x^n + 1$ enables multiplication to be done by negacyclic convolution, which can further be done by using point-wise multiplication in the NTT domain. The overall complexity of a single multiplication over the ring is then brought down from $O(n^2)$ to $O(n\log(n))$ .

## 2.4 HEAAN and Its Variant

Cheon et al. describe Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN) [CKKS17]. As noted, due to the ciphertext modulus requirement (prime or a power of two), it is difficult to represent the operation in RNS form. In 2018, Cheon et al. designed a HEAAN variant that allows for double-CRT representation and RNS-friendly modulus switching algorithms [CHK+18]. Applying approximate algorithms for modulus switching and careful selection of a fixed basis, this variant exploits the performance benefit of a full RNS implementation with a speedup factor of ten compared to the previous one. This variant is the first full RNS version of CKKS.

The HEAAN variant reduces the computation cost induced by large polynomial coefficients. Let $Q$ be a sufficiently large integer such that $Q = \prod_{l=0}^{L} q_i$. The CRT provides a ring isomorphism from $\mathbb{Z}_Q$ to $\prod_{l=0}^{L} Z_{q_i}$ when the primes in the basis $\{q_0, ..., q_L\}$ are coprime to each other. The small moduli $\{q_0, ..., q_L\}$ are chosen to satisfy $q_i/q_j \in (1-2^\eta, 1+2^\eta)$ so that noise level can be maintained in a specific level.

A set of special prime $\{p_0, ..., p_{k-1}\}$ is also chosen as an RNS base for key-switching operation. Let $P = p_0\, p_1 ... p_{k-1}$, then a ciphertext polynomial $c_1$ is firstly raised to $R_{PQ}$ to evaluate with key-switching keys and then reduced back to $R_Q$. The size of key-switching modulus $Q$ grows approximately linear with $\deg(R)$ so that CKKS can retain 128-quantum-bit security. The security benefit of having a larger $Q$ comes with a huge computational overhead (though there is a wide range of application-specific demands).

## 2.5 Generic RNS-CKKS

RNS-CKKS (CKKS) supports homomorphic operations, including multiplication (cipher-cipher, plaintext-cipher), addition, rotation, and conjugation. Two key oracles closely related to homomorphic multiplication are rescaling $RS(\cdot)$ and key-switching $KS(\cdot)$. Overall, encryption and decryption for CKKS can be summarized as follows [CHK+18].

*Encrytion*: Input {plaintext $m \in R_{Q_L}$, secret key $s \in R_{Q_L}$}; Uniformly sample $a \leftarrow U(R_{Q_L})$, $e \leftarrow \chi_{err}$, where $\chi$ is a specific error distribution, $ct = (ct_1, ct_2) = (-as+e+m, a)$; Output $\{ct\}$.

*Decryption*: Input {ciphertext $ct = (ct_1, ct_2)$, secret key $s \in R_{Q_L}$}; Calculate $m = ct_1 + ct_2 \cdot s$; Output $\{m\}$.

## 2.6 CASA

The proposed Compact and Scalable Architecture for Approximate Homomorphic Encryption (CASA) is built on the 2018 HEAAN variant from Cheon et al. [CHK+18] as it supports the core rounding operation that other approximate arithmetic HE schemes do not. While this CKKS is efficient for regression modelling, it is not so for other applications. The goal of CASA is that the accelerated CKKS can be practical for machine learning, artificial intelligence, and similar applications (especially in resource-constrained scenarios).

# 3 Primitive Function Modules

In this section, we give the details of the primitive function modules in CASA, along with related arithmetic innovations and optimization techniques.

## 3.1 Modular Reduction Module

Polynomial multiplication is one of the most expensive operations required when working with rings and fields. Usually, for NTT-friendly rings such as $\mathcal{R} = \mathbb{Z}_q[x]/(x^n+1)$, reduction with respect to the modulus polynomial is optimized and only modular multiplication with respect to the coefficients of the polynomials is left. Various approaches have been proposed for optimizing the NTT by considering NTT-friendly and unfriendly rings together with known techniques, as well as exploring possible optimizations of the underlying arithmetic using special primes.

The focus of our work in this respect is how to optimize the NTT itself (butterflies) as well as the modular multiplication and reduction with respect to the RNS coefficients. We have carefully considered the features of the existing reduction algorithms and the moduli-chain requirement in CKKS and decided to propose a new partially reduction-free strategy for CASA, including a new algorithm and a fine-tuned hardware architecture.

### 3.1.1 Brief Background

Modular multiplication comprises two main operations: multiplication and reduction. Such multipliers can be implemented in an interleaved fashion or separate scanning. Proper selection of an algorithm is done according to the operand size and the field, and their applicability depends on whether the modulus is special or generic. These algorithms are classified by the direction in which they apply the modular reduction to a product: from the right, from the left, or from both ends in parallel [KAK96]. Primary examples of these are Montgomery [Mon85], Blakely [Bla83], and BMM (bipartite modular multiplication) [KT05], respectively.

### 3.1.2 Existing Multiplication and Reduction Algorithms

Montgomery is the most widely applied algorithm for multiplication, reduction, and interleaved multiplication and reduction. If the multiplication and reduction are split, the reduction routine is usually paired with a multiplication algorithm, such as Karatsuba or Schönhage–Strassen depending on the application [KA98, KO63, SBE15, SCH71]. Barrett [Bar00] and Longa and Naerig [LN16] are the most common reduction algorithms. Barrett optimizes the division part of modular reduction by replacing it with multiplication and computing the residue through approximation of the quotient [Bar00]. Longa and Naerig is an optimized reduction algorithm that applies when the modulus is a prime of special form [LN16], i.e., one must find primes that are compatible with the CKKS and Longa and Naerig reduction, while ensuring that such primes preserve the security of CKKS. Barrett

and Montgomery are commonly selected for reduction (with preference for Montgomery when it concerns hardware implementations) [PS21].

### 3.1.3 Interleaved Algorithms

In general, interleaving is complicated. Blakely and Montgomery are two well-known interleaved algorithms [Mon85, Bla83]. Though such algorithms might seem attractive because CKKS deals with a chain of different prime moduli, a generic algorithm would be more desirable. Thus, investigating a more efficient algorithm that can conform to a given FHE scheme (CKKS) is worth pursuing since the performance gain can be significant.

### 3.1.4 Recent Advances

Our focus is on bipartite algorithms that can perform interleaved multiplication and reduction in parallel from both ends. The first such algorithm is BMM [KT05] followed by a radix-4 version in [KT08]. This was followed by similar works, such as the partially interleaved bipartite multiplier from Saldamlı et al. using Karatasuba multiplication (also known as Karatsuba-Ofman, KO) [SBE15]. More interestingly, is the reduction-free fully interleaved Karatsuba-Ofman modular multiplier, called RF-FIKO proposed in [OMSL+23]. RF-FIKO presents an advantage over BMM by exploiting the special form of the modulus (a reduction-free trinomial or RFT) to eliminate reduction circuits. Overall, RF-FIKO is a special case of the fully interleaved modular multiplication that integrates Karatsuba multiplication with bipartite reduction.

### 3.1.5 Prime Number Selection.

The CKKS requirements for the primes that form the chain determine which modular multiplication algorithms can be used. To further reduce the area consumption, we exploit the similarity among different CKKS moduli. A first thought on optimization would be using a shift-and-add strategy to take advantage of sparing '1's in the selected primes. Related prime representation and search methods include Canonical Signed Digit (CSD) recoding, which is a method for finding sparse primes and representing them in an even more spare form using "signed" representation of power-of-two values [DIZ07]. The work of [PS21] used the CSD representation and proposed an FPGA-based NTT accelerator and evaluated for polynomial degrees of 1,024 and 4,096 at coefficient bit-width of 28 and 30. However, practical implementations of CKKS usually require larger primes with bit-width ranging from 54 to 60 bits and which lie in a narrow range compliant with the form of Proth primes [AS10].

In this paper, the prime number selection for CKKS is carried out as follows. First, we search for Proth primes at each bit-width number ranging from 54 to 60. Then, we create a group for each prime $p_i$ consisting of all Proth numbers $p_j$ we have already acquired such that $q_i/q_j \in (1-2^{\eta}, 1+2^{\eta})$ to satisfy the CKKS requirements for noise growth [CHK+18]. Since the bit precision $\eta$ is correlated closely with the rescaling precision, [CHK+18] listed several valid prime numbers for 55-bit sized primes in Appendix B to elaborate prime selection. The $\eta$ was chosen to be 31 which is more than half of the prime bit length. This value of $\eta$ allowed for computing the same ciphertext as in HEAAN, meaning rescaling accuracy was not impacted. The CKKS requirements for the primes do not conform to an RFT form to apply RF-FIKO. We explore primes for which the Blakely circuit can be removed and the Montgomery circuit remains but with minimal modifications in the existing operations. We found several primes of size 54 to 60-bit satisfying a partially reduction-free variant of RF-FIKO by using the program open sourced by [CHK+18] to generate candidate prime numbers for this step. The next step is to determine $L$ prime numbers within each group. The target for this process is set to find the group for the $L$

primes that have the least possible bits set. We take a brute-force approach to implement this process because the number of possible primes is very limited. Finally, we compare the best list within each group to acquire $L$ primes for our prime chain and design a fine-tuned micro-architecture to do the modular multiplication (details are described below). We selected nine 54-bit primes where $\eta \in [19, 23]$ and hence, the impact on rescaling accuracy remains the same as in [CHK+18].

### 3.1.6 Novel Partially Reduction-Free Modular Multiplier

**Limitation of the Existing Method.** First, we would like to highlight the origins of PRF-FIKO starting from the first bipartite algorithm [KT05]. For simplicity, we keep this subsection and the description of the new algorithm in $GF(2^n)$. BMM computes the bipartite residue of $a \times b \pmod{F(t)}$ as follows. It takes in two operands, splits one into approximately equal halves, and solves the problem in two parts that can be computed in parallel. For example, consider two operands $a = \{a_1, a_0\}$ and $b = \{b_1, b_0\}$, where the subscript 1 denotes the upper word of an operand and the subscript 0 the lower word. Note we use this subscript convention throughout except for the Karatsuba terms where the appended subscript $h$ and $l$ would refer to an upper or lower word, but the 0 or 1 is part of the term label. Then BMM will compute a residue $s = a \times b_1 \pmod{F(t)}$ and $t = a \times b_0 \pmod{F(t)}$ using Blakely and Montgomery interleaved multiplication and reduction algorithms in parallel, where F(t) can be split as two polynomials $P = \{f_1, f_0\}$ to represent the prime modulus. At the end, it will sum $s + t + F(t)$ to obtain the bipartite residue.

PIKO is a bipartite modular multiplier that separates the multiplication and reduction parts [SBE15]. Karatsuba multiplication is applied using the first level of recursion (with half-size words) to compute a product, it is partial and not fully interleaved multiplication. Montgomery and Blakely reduction algorithms are applied to two of the Karatsuba terms, $c_0 = a_0 \times b_0$ and $c_2 = a_1 \times b_1$, to obtain their quotients $\pmod{f_0}$ and $\pmod{f_1}$, respectively. Each of the three Karatsuba terms, including the cross-term $c_1 = (a_1 + a_0)(b_1 + b_0)$, is then summed with a respective bipartite term. Note, naming of the Karatsuba terms varies and we follow the definition used here. Lastly, a bipartite residue is computed as a final sum of the updated Karatsuba terms.

RF-FIKO is an improvement to PIKO in two respects. First, it eliminates the reduction circuits completely, meaning that the Montgomery and Blakely reduction algorithms are no longer needed. This result was obtained by exploring improvements to the form of the modulus polynomial, which would allow the elimination of such circuits used by PIKO for the purpose of computing the Montgomery and Blakely quotients. RF-FIKO was developed considering desired forms of the modulus polynomial, the implications on the computations, and exploring the possibilities when working with half-size words. Experimenting with bit by bit computations in $GF(2^n)$, a known fact can be easily observed–that if the lower half of $F(t)$ is only 1, the Montgomery quotient of any input $a$ reduced by 1 is simply $a$ and hence, the Montgomery reduction circuit can be eliminated. Similar observations with respect to the upper half of $F(t)$ allowed the elimination of the Blakely reduction circuit.

For cryptographic purposes, we require working with an irreducible polynomial of odd degrees, preferably a trinomial or pentanomial. These observations collectively led to defining $F(t)$ as a trinomial where the upper word is a binomial and the lower word is a 1. Such a trinomial is referred to as an RFT in [OMSL+23]. This means the Montgomery and Blakely quotients can be taken directly from the Karatsuba terms $c_0$ and $c_2$ when $F(t)$ is an RFT. More specifically, the Montgomery quotient is found in the lower word of $c_0$ and the Blakely quotient in the upper word of $c_2$ starting from the second most significant bit of $c_2$ since the operands are prefixed to match the bit-length of $F(t)$ for symmetry. Second, it was observed that both PIKO and RF-FIKO can be implemented in a fully-interleaved fashion. The two improvements: removing reduction circuits and

fully interleaving the computations simplified PIKO to RF-FIKO where the core of the algorithm consists of only three multiplications followed by some additions to compute the final sum or bipartite residue.

The RF-FIKO was developed and tested for the operation over binary basis [OMSL+23], but has not been explored further to be compatible with actual FHE schemes. Meanwhile, the presented approach was still in the early stage, i.e., only produces one bit per cycle and it is limited by a fixed modulus.

**Our Proposal.** We observe that RF-FIKO solves the problem of modular multiplication and reduction by obtaining a bipartite residue with three simple half-size word multiplications and some shifts and additions. This algorithm applies when the modulus polynomial has the special RFT form that can give us reduction-free quotients. Because the CKKS requirements for the primes clearly do not conform to an RFT form, we consider a search for primes for which either the Blakely or Montgomery circuit can be removed. Specifically primes where either the upper word has only the most significant bit set or the lower word has only the least significant bit set (is a 1) to develop a variant of RF-FIKO with minimal changes.

Based on this background, we propose to: (i) explore RF-FIKO further by testing for generation of CKKS primes with RFT form; (ii) if the primes are not in the RFT form, then find primes for which we can have a partially reduction-free algorithm. If (i) or (ii) is successful, the algorithm can be integrated as a replacement for modular multiplication and reduction; (iii) develop a low-latency parallel algorithm that requires no sequential data flow control, so that the point-wise multiplication and reduction can be finished in one cycle; (iv) design a fine-tuned hardware micro-architecture for the parallel algorithm with minimum resource usage.

**Algorithm 1** Partially Reduction-Free Fully Interleaved Karatsuba (PRF-FIKO) Modular Multiplication Algorithm

**Input**: $a$, $b \in P$ s.t. $P$ is represented as an array of indices
**Output** : $c \equiv a \times b \times 2^{-r} \pmod{P}$
$c_0 \leftarrow a_0 \times b_0$ ▷ terms from Karatsuba Algorithm
$c_1 \leftarrow (a_1 + a_0)(b_1 + b_0)$
$c_2 \leftarrow a_1 \times b_1$
$q_1 \leftarrow c_2 \leftarrow$ Blakely quotient
$q_0' \leftarrow \text{Montgomery}(c_0, f_0)$ ▷ apply Montgomery reduction to $c_0$ with $f_0$
$t_0 \leftarrow \text{MMSM}(q_0', f_0)$ ▷ BMM terms, computes $q_0' \times f_0$
$t_2 \leftarrow \text{MMSM}(q_1, f_1)$ ▷ computes $q_1 \times f_1$
$t_1 \leftarrow \text{MMSM}(q_0' + q_1, f_0 + f_1)$ ▷ cross-term uses sparse-multiplier internally
$c_0' \leftarrow c_0 + t_0$ ▷ add the bipartite reduction terms
$c_1' \leftarrow c_1 + t_1$
$c_2' \leftarrow c_2 + t_2$
$c \leftarrow (c_{0h}' || c_{2l}') + (c_{2l}' || c_{0h}') + c_1'$
**Return** $c$

**Proposed Algorithm.** CKKS requires a prime modulus chain of size $L$, which grows approximately linear to $n$. Generating CKKS compatible primes with RFT form is not feasible. However, several primes can be generated for which the most significant bit is set and the bits in the lower word are sparse. Such primes allow for a partial variant of RF-FIKO. The proposed new partially reduction-free algorithm called Partially Reduction-Free Fully Interleaved Karatsuba (PRF-FIKO) modular multiplier, is shown in Algorithm 1. Overall, we have improved the original algorithm of RF-FIKO in two aspects: flexibility with the modulus polynomial $F(t)$ and bipartite reduction. The original algorithm computes the Karatsuba terms and makes use of the known special form of

$F(t)$ to eliminate reduction circuits and implement remaining multiplications as shifts and additions. PRF-FIKO re-adds the Montgomery reduction circuit and requires a minimal sparse multiplier for the bipartite cross-term.

Step-by-step, Algorithm 1 computes the bipartite residue as follows. Let the prime $P$ be represented in the binary basis as a modulus polynomial $F(t)$ that defines a finite field $GF(2^n)$. Lines 3-5 compute the Karatsuba terms. Line 6 takes the Blakely quotient from $c_2$ since these are directly found in the uppermost significant bits of $c_2$. By virtue of how we compute the Blakely quotient bits given the form of $f_1$, which has only the most significant bit set, we can easily observe those bits are just the $c_2$ bits. Moreover, we are only interested in the upper word of the Blakely quotient and the lower word of the Montgomery quotient, since we are working with half-size words. Now, because the modulus polynomial is no longer an RFT, the lower bits in this case vary, meaning the bits in $f_0$ vary and line 7 computes the Montgomery quotient $q'_0$ obtained from reducing $c_0$ with $f_0$. Lines 8-10 compute the bipartite terms $q'_0 \times f_0$, $q_1 \times f_1$, and $(q'_0 + q_1) \times (f_0 + f_1)$. Because $F(t)$ is no longer an RFT, multiplication for the bipartite cross term $(q'_0 + q_1) \times (f_0 + f_1)$ requires a sparse multiplier since multiplication by $(f_0 + f_1)$ is no longer a simple left shift by a fixed power of two. Lines 11-13 represent partial products by adding the bipartite terms to the respective Karatsuba terms. Note, the bipartite terms may also be referred to as "bipartite reduction terms" since they reduce the Karatsuba terms. Lastly, line 14 computes the bipartite residue as a sum of the reduced terms. This is in line with the desired residue of $a \times b \times 2^{-h} \pmod{F(t)}$ where $h = k/2$ denotes the bit length of a half-size word and $k$ the bit length of the $F(t)$. A complete derivation of the Karatsuba-Ofman Algorithm shows that the desired bipartite residue for the product modulo $F(t)$ is obtained by adding to $c'_1$ the partial products $(c'_0 + c'_0 2^h + c'_2 + c'_2 2^h)$ [KO63]. Note that because $c'_0$ and $c'_2$ have been reduced by their respective bipartite terms, we can discard the lower word of $c'_0$ and the upper word of $c'_2$. This means that the remaining additions $(c'_0 + c'_0 2^h + c'_2 + c'_2 2^h)$ to $c'_1$ can be expressed as in line 14 where $||$ denotes concatenation. Lines 3-5, 7, and 8 can be computed in parallel. Line 9 is multiplication by $f_1$ and can be implemented as a simple left shift by the power of the degree of $f_1$. Line 10 takes in the computed $q'_0$ term to compute the cross term, and finally, lines 11-14 (including Line 9) can be combined to compute the bipartite residue. Naturally, the same algorithm can be implemented in a single cycle.

In PRF-FIKO, the modulus polynomial $F(t)$ now takes the form of the CKKS selected primes. The upper word has the most significant bit set and the lower word varies with two to six bits set. This allows us to use an array of indices to represent selected primes instead of requiring the entire value of a given prime or $F(t)$. For example, if $F(t)$ has degree 163, the RF-FIKO takes in a 164-bit array. In PRF-FIKO, the selected primes we use can be represented with at least two and at most six bits. Hence, we let $F(t)$ be an array of indices that can represent any of our selected primes. Moreover, because the arithmetic for the BMM terms involves shifts and additions by known amounts due to the known form of $F(t)$, we further optimize this part by incorporating a Multi-Modulus Shift Multiplier (MMSM) that performs the BMM steps for a given modulus based on the bits set of the 6-bit array $F(t)$. The difference here is that the cross-term uses a sparse multiplier since the lower word of $F(t)$ is no longer one.

**Fine-Tuned Hardware Micro-architecture.** We present a novel partially reduction-free hardware micro-architecture with significant performance enhancement by studying the relationship between different prime moduli and exploring hardware resource multiplexing. The selected $L$ moduli form a moduli chain as mentioned in the previous subsections. We can write all the moduli in binary format and collect all the positions for '1's, e.g., suppose we have six different positions, we can index them from 0 to 5, as seen from Fig. 2. The first micro-architecture we construct is the multi-modulus shift multiplier (MMSM). When a key-switching, rescaling, mod-up or mod-down operation is executed, a modulus is

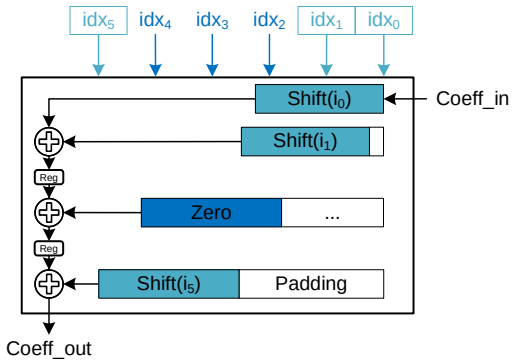



**Figure 1:** Multi-Modulus Shift Multiplier (MMSM), where idx refers to index.

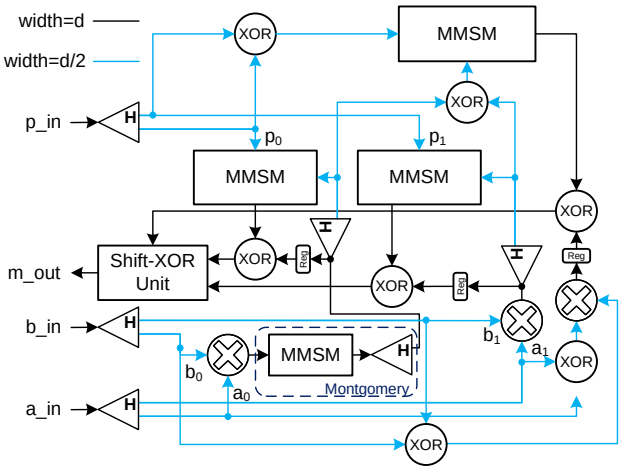



**Figure 2:** Details of the proposed partially reduction-free module. Subscript H refers to the high word of a given register.

determined and configured by setting the corresponding index to 1. Such an architecture is suitable for FPGA-specific optimization since it creates clear critical paths between arithmetic components. As shown in Fig. 1, inserting registers between adders allows a higher clock frequency at the cost of sequential components. On the other hand, as shown in Fig. 2, the delay incurred by inserting registers into MMSM has to be compensated by inserting register arrays into several data paths to ensure correct data capture at required clock cycles. Additionally, our investigation extends to optimizing digital signal processing (DSP) aspects. For instance, we can further enhance the clock frequency when working with Xilinx Artix-7 devices equipped with onboard DSP48E1 units supporting 25-bit $\times$ 18-bit multiplication. On top of the previously mentioned strategy of inserting registers, we divide the operands into distinct bitstring groups, each with a size of less than 18 bits. In this scenario, registers are inserted between these bitstring groups rather than at various layers of adders.

**Complexity and Discussion.** In this way, we construct a fine-tuned micro-architecture for a low-latency, modulus changeable, and partially reduction-free multiplier. The complexity analysis of the proposed design is as follows. Direct multiplication of two polynomials in a ring with coefficients defined over $\mathbb{Z}_q$ has a complexity of $O(n^2)$ with lazy reduction. The NTT method brings this complexity to $O(n\log n)$. With respect to the polynomial coefficients, standard Barrett [Bar00], Blakely [Bla83], and Montgomery [Mon85] have

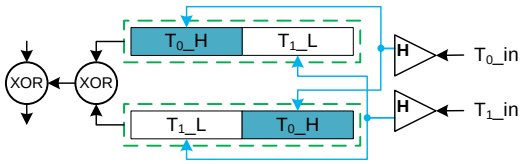



**Figure 3:** Shift-XOR Unit. Subscripts H and L refer to the high and low words of a given register, respectively.

complexity $O(n^2)$ for $n$-digit operands. Bit-parallel interleaved algorithms in binary polynomial basis for Blakely [Bla83] and Montgomery [Mon85] would have $O(n)$ complexity. BMM [KT05], RF-FIKO [OMSL+23], and PRF-FIKO have timing proportional to $n/2$. Additional improvements for PRF-FIKO are in the form of optimization techniques and further exploration of the underlying arithmetic. For example, a speedup can be incrementally doubled at the cost of space. The recursion depth can also be increased at the cost of added complexity.

Fig. 2 is a combinational circuit based architecture, whereas the existing approaches (such as RF-FIKO) are sequential circuit based structures. Therefore, an analysis with respect to time and space complexities can be better observed in the top-level module that incorporates MSMM (such as NTT and Dyadic & Accumulation), i.e., our approach obtains significant improvement in both complexities. The proposed architecture can be efficiently switched among different prime moduli for CKKS, whereas the existing methods did not offer this property. For instance, when $n = 2^{14}$ and the moduli chain contains 8 primes, the proposed multiplier can shift between 8 moduli. A straightforward way to implement the existing methods would need to construct 8 different modular components. Furthermore, the proposed partially reduction-free modular multiplier integrates the multiplication and reduction parts together. Thus, a comparison by direct programming and synthesizing between the proposed multiplier and modular structure and the existing methods would not be enough. Nevertheless, both the analysis here and the final implementation (see Section 5) confirm the efficiency of the proposed partially reduction-free technique.

**Algorithm 2** NTT Algorithm

**Input**: Polynomial $x = (x_0, ..., x_{n-1}) \in R_p$, modulus $p$
**Output**: NTT of $x$
**Initialize**: Primitive $n$-th root of unity $\omega_n$
**for** $i \leftarrow \log_2 n$ downto 1 **do**
  $m \leftarrow 2^i$
  $W \leftarrow \omega_n^{n/m} \pmod p$
  **for** $j \leftarrow 0$ *to* $n-1$ **do**
    $W' \leftarrow 1$
    **for** $k \leftarrow 0$ *to* $m/2-1$ **do**
      $t \leftarrow x[j+k]$
      $u \leftarrow x[j+k+m/2]$
      $X \leftarrow t+u$
      $Y \leftarrow (t-u)W'$
      $x[j+k] \leftarrow X \pmod p$
      $x[j+k+m/2] \leftarrow Y \pmod p$
      $W' = W'W \pmod p$
    **end for**
  **end for**
**end for**
Return x

## 3.2 NTT Module

NTT operation (Algorithm 2) is not as efficient on CPU/GPU as on FPGA due to an FPGA's high parallelism processing style. Extensive studies have been conducted for NTT over different parameter sets, such as coefficient modulus size, input polynomial size, and the total execution time in the targeted application [ZLL+21, KLC+20, DNKYL22]. FPGA accelerators for FHE implementations also contribute a variety of ways for constructing the NTT module [RLPD20, HZL+22, MKS+22, TRV20, NSA+22]. We studied these literature works and have presented an in-place NTT architecture for compact and scalable CKKS accelerator (Fig. 2). We use parameter $N_{bf}$ as a design parameter not only for representing the number of butterfly cores in an NTT module but also directly relate it to design parameters for other modules. In this perspective, we create a one-variable configuration for the entire accelerator. The performance of the entire accelerator with respect to different scalable factors is also categorized by this parameter (Section 5).

Let $N_{bf}$ denote the number of butterflies we use in parallel. The first question is how we organize memory for data storage. This is closely related to butterfly data flow and NTT stages. An NTT operation consists of $\log_2 n$ stages requires each stage to operate each coefficient once. [RLPD20] proposes to divide these stages into two types. The first type (Type-I), consists of the first $(\log_2 n - \log N_{bf} - 1)$ stages (indexed from 0), features the characteristic that the coefficients of butterfly inputs $X_i$ and $Y_i$ distant from each other by a relatively large distance, or precisely the distance greater than $2N_{bf}$. The second type (Type-II), then, consists of the rest stages. This method requires a MUX-based architecture to route coefficients within the same memory address location to the desired butterfly's inputs and therefore contributes to a considerably large logical circuit consumption. Another method allowing multiple butterflies to work in parallel and achieved an excellent performance was proposed by [PS21]. This method requires $N_{bf}$ to be a power-of-two, the same as the aforementioned processing style. It then divides the polynomial into even and odd parts, which fits well with the proposed memory accessing pattern for data flow routing. Besides that, we do not need a MUX-based architecture to route between different butterfly cores. However, this processing style needs a doubled memory to store all $n$ points to satisfy the proposed access pattern.

Since we want to design a compact accelerator and memory read/write contributes a lot to FHE when deploying in many practical applications, we just follow [RLPD20] to divide the NTT stage into Type-I and Type-II. Our memory bank architecture therefore can be organized as shown in Fig. 4. We instantiate at least $2N_{bf}$ on-chip memories to acquire $2N_{bf}$ memory ports. Note that more than one block memory tiles are possibly used in one memory element.

The type-I stages require $X_i$ and $Y_i$ fetched from different memory addresses, which incurs an inter-memory-address dependency. This memory access pattern for these stages is implemented by using a hybrid of finite-state-machine (FSM) and several address tables. First, we generate address tables to indicate the step width between $X_i$ and $Y_i$ for each stage. Then, we generate the table for each stage to indicate whenever a jump is needed within a stage. Finally, the FSM generates control signals for address management with the help of the address tables. The address tables and FSM combined to form the address logic unit in Fig. 4. The type-II stages fetch/store $X_i$ and $Y_i$ from the same memory address due to the relatively small step width. This stage requires a MUX-based combinational circuit structure to route $X_i$ and $Y_i$ to the desired butterflies. [RLPD20] suffered from the same issue and proposed to optimize this huge MUX system by eliminating some small MUX at a fixed location, e.g. the index 0 butterfly's input $X$ always does not need location change. However, this method still requires a large MUX-based circuit, according to our evaluation. Thus, we propose to construct the MUX circuit by utilizing the observation that only $\log N_{bf}$ stages need routing. These stages include stage $\log_2 n - \log N_{bf}$ to stage $\log_2 n - 2$ and the last stage (stage $\log_2 n - 1$) does not need coefficient routing. Due to the

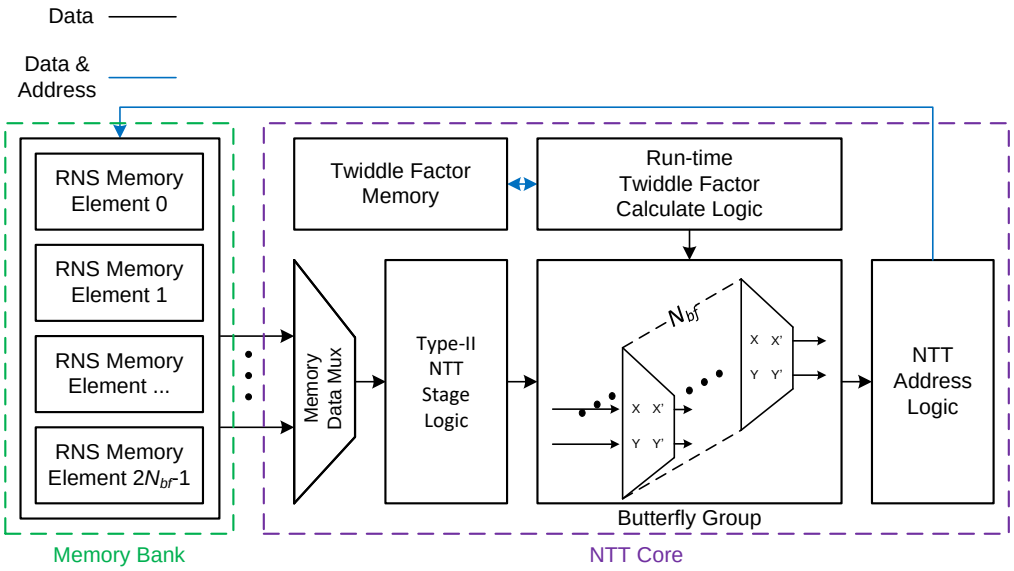



**Figure 4:** Proposed in-place NTT architecture for polynomials represented in RNS format.

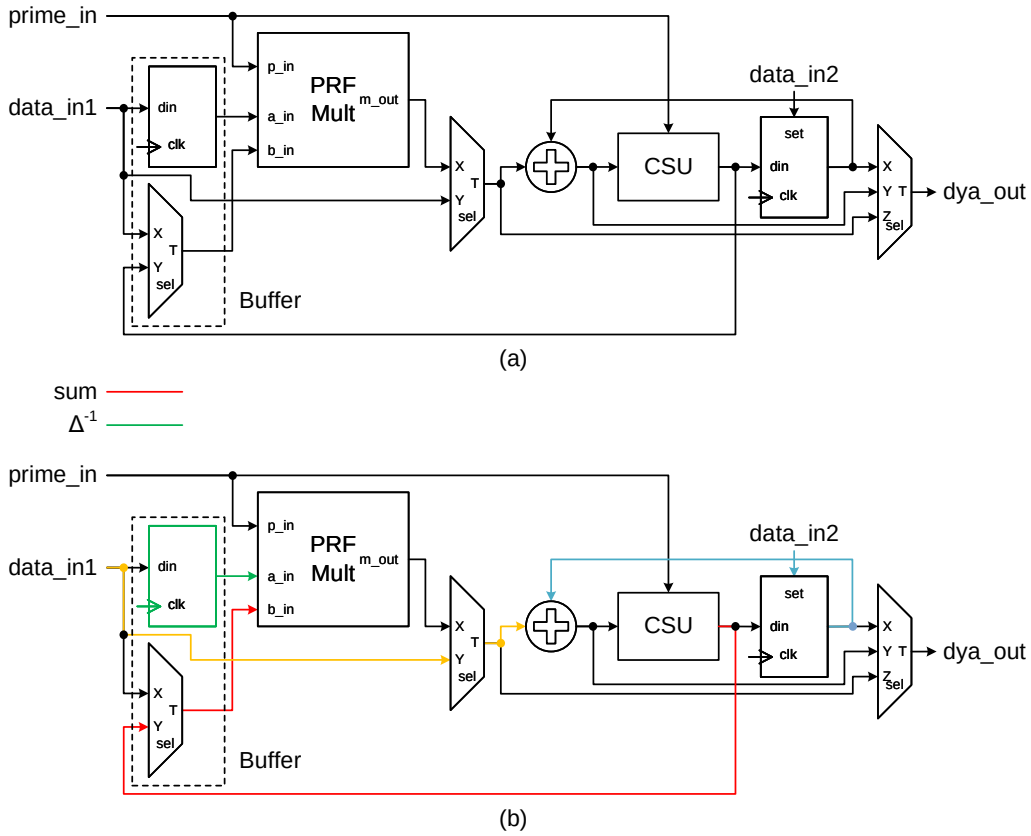
fact that: (a) the logarithm complexity of the number of stages needs routing with respect to $N_{bf}$ and (b) $N_{bf}$ is intended to be small in a compatible design, we can efficiently reduce the area consumption for this MUX-related logic. For instance, when $N_{bf} = 8$, type-II stages require only 3 different patterns. We can use a 3-to-1 MUX for each bit to execute this task. These MUXes are packed into the Type-II NTT Stage Logic Unit (Fig. 4). We also use decimation in frequency (DIF) strategy for NTT and decimation in time (DIT) strategy for INTT in this design. The DIF NTT is shown in Algorithm 2. The twiddle factors are stored in the Twiddle Factor Memory, which is a dual-port memory of depth 2 (address space ranges from 0 to 1). When the root for NTT is determined and stored at address 0, the rest of the roots used in the entire NTT process is calculated dynamically. The same strategy is used for Type-I address logic. We store multiplicand in address 1 and update at running time. The multiplicand is multiplied with the last cycle's roots to derive roots for the current cycle.

**Algorithm 3** Homomorphic Multiplication Algorithm

**Input**: Ciphertext $\hat{c} = (\hat{c_0}, \hat{c_1}), \hat{c'} = (\hat{c'_0}, \hat{c'_1}) \in R^2_{Q_l}$ decryptable with secret $(1, s)$
**Output**: Ciphertext $d = (\hat{d}_0, \hat{d}_1, \hat{d}_2) \in R^3_{Q_l}$ decryptable with secret $(1, s, s^2)$
**for** $i \leftarrow 0$ *to* $l$ **do**
    $\hat{d_0}\{i\} \leftarrow \hat{c_0}\{i\} \otimes \hat{c'_0}\{i\}$
    $\hat{d_1}\{i\} \leftarrow \hat{c_0}\{i\} \otimes \hat{c'_1}\{i\}$
    $\hat{d_1}\{i\} \leftarrow \hat{d_1}\{i\} + \hat{c'_0}\{i\} \otimes \hat{c_1}\{i\}$
    $\hat{d_2}\{i\} \leftarrow \hat{c_1}\{i\} \otimes \hat{c'_1}\{i\}$
**end for**
Return $d = (\hat{d}_0, \hat{d}_1, \hat{d}_2)$

## 3.3 Dyadic and Accumulation (ACC) Module

The Dyadic and ACC Module is shown in Fig. 5. Dyadic multiplication is used in homomorphic multiplication, key-switching, and rescaling algorithms, as shown in Algorithm 3,




. CSU: coefficient subtraction unit.

**Figure 5:** (a) Architecture of Dyadic & ACC Module. (b) Data flow for Rescaling operation. PRF Mult: partially reduction-free multiplier

Algorithm 4, and Algorithm 5 respectively. We design a highly efficient micro-architecture for Dyadic and ACC Module by multiplexing hardware resources. The Dyadic and ACC Module eventually embeds 4 functions into one structure including dyadic multiplication, dyadic multiplication and accumulation, addition, and rescaling. Firstly, we instantiate $N_{bf}$ point-wise multipliers (PRF Mult in Fig. 5) using the proposed partially reduction-free method (in Section 3.1) to reduce the area occupation. For the Dyadic multiplication operation, a straight-forward data flow is executed, i.e., operands are fed into PRF Mult by data_n1 one cycle by one cycle. Then, the 2-to-1 MUX before the adder selects input X. It takes 2n-cycles to multiply two size $n$ polynomials with one Dyadic core in this way.

For accumulation operation, the coefficient subtraction unit (CSU) is responsible for subtracting one fold of modulus $p$ from the sum if the sum is larger than $p$. We also implement the polynomial addition in this module by multiplexing the adder. Data_in2 is connected to the adder by using the set port of register so that one coefficient goes into the adder from the output of the register (another from the input buffer). When rescaling operation is executed, Line 9 Algorithm 5 indicates $\Delta^{-1}$ is multiplied by the sum of $\hat{c_1}$ and $\hat{t_1}$ (this data flow is shown in Fig. 5 (b)). Again, we multiplex the adder by selecting one coefficient from the output of the register and another from data_in1. Then the sum is multiplied by $\Delta^{-1}$ (pre-stored in the register). Note here we also multiplex the register of the input buffer.

**Algorithm 4** Key-Switching Algorithm

**Input**: Ciphertext $d = (\hat{d}_0, \hat{d}_1, \hat{d}_2) \in R^3_{Q_l}$ decryptable with secret $(1, s, s^2)$
**Output**: Ciphertext $\hat{c'} = (\hat{c'_0}, \hat{c'_1}) \in R^2_{Q_l}$ decryptable with secret $(1, s)$ $\hat{d'_2} \leftarrow \text{ModUp}(\hat{d}_2)$
**for** $i \leftarrow 0$ *to* $l + k$ **do**
    $\hat{t_0}\{i\} \leftarrow \hat{d'_2}\{i\} \otimes swk_0\{i\} \bmod q_i$
    $\hat{t_1}\{i\} \leftarrow \hat{d'_2}\{i\} \otimes swk_1\{i\} \bmod q_i$
**end for**
$\hat{t'_0} \leftarrow \text{ModDown}(\hat{t_0})$
$\hat{t'_1} \leftarrow \text{ModDown}(\hat{t_1})$
**for** $i \leftarrow 0$ *to* $l$ **do**
    $\hat{c''_0}\{i\} \leftarrow \hat{d_0}\{i\} + \hat{t'_0}\{i\}$
    $\hat{c''_1}\{i\} \leftarrow \hat{d_1}\{i\} + \hat{t'_1}\{i\}$
**end for**
Return $\hat{c''} = (\hat{c''_0}, \hat{c''_1})$

**Algorithm 5** Rescaling Algorithm

**Input**: Ciphertext $\hat{c} = (\hat{c_0}, \hat{c_1}) \in R^2_{Q_l}$ encrypting message $\Delta z$
**Output**: Ciphertext $\hat{c'} = (\hat{c'_0}, \hat{c'_1}) \in R^2_{Q_{l-1}}$ approximately encrypting message $z$
$c_0\{l\} \leftarrow \text{INTT}(\hat{c_0}\{l\}, q_l)$
$c_1\{l\} \leftarrow \text{INTT}(\hat{c_1}\{l\}, q_l)$
**for** $i \leftarrow 0$ *to* $l - 1$ **do**
    $\hat{t_0} \leftarrow \text{NTT}(-c_0\{i\}, q_i)$
    $\hat{c'_0} \leftarrow \Delta^{-1}(\hat{c_0} + \hat{t_0}) \bmod q_i$
    $\hat{t_1} \leftarrow \text{NTT}(-c_1\{i\}, q_i)$
    $\hat{c'_1} \leftarrow \Delta^{-1}(\hat{c_1} + \hat{t_1}) \bmod q_i$
**end for**
Return $\hat{c'} = (\hat{c'_0}, \hat{c'_1})$

## 3.4 Memory Organization

Memory related data movement is considered as a major bottleneck for CKKS hardware design [SFK+21, SFK+22]. The memory system for a specific-domain accelerator such as a CKKS accelerator can be categorized into on-chip memory inclined style [MKS+22] or on-off memory hybrid style [HZL+22, RLPD20]. When $n = 2^{14}$, each coefficient of polynomial before decomposition is at least 438 bits. This results in a considerably large ciphertext size at 13.69MB which is 106.7% of the on-chip memory (Block memory or BRAM) for a resource-constrained device like Artix-7. It can be challenging even for advanced FPGAs such as the data-center FPGA Alevo U250. While [MKS+22] relied all on Alevo U250, other high-speed designs with cascaded [RLPD20] or ring-like [MKS+22] structures had to use the off-chip memory to store the data (even their FPGAs were very advanced). CASA takes a contrary way of organizing the memory architecture. We argue that for scalable memory utilization, CKKS accelerator should incline to use the off-chip memory as much as possible to keep the on-chip structural simplicity and straightforwardness (while on-chip memory can directly connect to the other modules). With a large amount of data stored on the chip, the address space and port fanout then need to be enlarged for parallel computation. The on-chip address management thus requires adding more control logic, which may result in a more complicated data exchange process with external memory. On the contrary, relying on off-chip memory is relatively

easier. Data can be read into/out of the on-chip memory in trunk by manipulating a memory offset variable since off-chip memory has a significantly larger memory space.

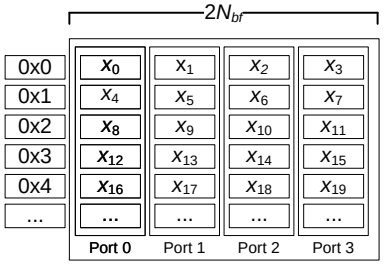



**Figure 6:** Memory organization of the Memory Bank for $N_{bf}$=2. $x_i$ stands for the $i$-th coefficient in an polynomial.

**Algorithm 6** ModUp Algorithm

**Input**: Ciphertext component $\hat{c} = (c\{0\}, ..., c\{l\}) \in R_{Q_l}$
**Output**: Ciphertext component with special primes $\hat{c'} = (c\{0\}', ..., c\{l+k\}') \in R_{P_lQ_k}$
**for** $i \leftarrow 0$ *to* $l$ **do**
    $c\{i\} \leftarrow \text{INTT}(\hat{c}\{i\}, q_i)$
    $c\{i\} \leftarrow c\{i\} \otimes [\hat{q}_i^{-1}]_{q_i}$
**end for**
**for** $j \leftarrow 0$ *to* $k-1$ **do**
    $s \leftarrow 0$
    **for** $i \leftarrow 0$ *to* $l$ **do**
        $s \leftarrow s + c\{i\} \otimes [\hat{q}_i]_{p_j} \bmod p_j$
    **end for**
    $c\{l+j+1\}' \leftarrow s$
    $\hat{c}\{l+j+1\} \leftarrow \text{NTT}(c\{l+j+1\}', p_j)$
**end for**
Return $\hat{c'} = (c\{0\}', ..., c\{l+k\}')$

The proposed memory bank is described in Fig. 6. We have used two memory banks as the on-chip memory to provide data for all three computation components, namely the NTT Module, the Dyadic and Accumulation (ACC) Module, and Modular Switching Bank. Since we only use one NTT Module (as shown later in Fig. 7), the requirement for maximum data transfer bandwidth between off-chip memory and on-chip memory can be achieved even on the resource-constrained device at this level of throughput rate. For example, for parameter set III of CKKS and RNS prime width set to 54 bits, an NTT module works on 884,736 bits of data trunk (one polynomial) for 14,436 cycles. If the programmable logic part works at 100MHz, it will require 71.68$\mu s$ to finish the transform of a polynomial. In this case, even without using PCIe, the AXI-4 interface working at 50MHz and 1024-bit width will be able to transfer 3,670,016 bits of data. This is also an important factor that supports a memory design leaning on off-chip memory. We have also demonstrated the detailed memory usage for CASA in Tables 2 and 4.

## 3.5 Modulus Switching Bank

Modular switching operation is a basis conversion operation that maps a ciphertext from modulus $P_l$ to a larger modulus $P_lQ_k$, or in reverse [CHK+18]. The modulus switching algorithm for conversion from $P_l$ to $P_lQ_k$ is shown in Algorithm 6. The dyadic multiplication

**Algorithm 7** ModDown Algorithm

**Input**: Ciphertext component with special primes $\hat{c} = (c\{0\}, ..., c\{l+k\}) \in R_{P_l Q_k}$
**Output**: Ciphertext component $\hat{c}' = (c\{0\}', ..., c\{l\}') \in R_{Q_l}$
**for** $j \leftarrow 0$ *to* $k-1$ **do**
    $c\{l+j+1\} \leftarrow \text{INTT}(\hat{c}\{l+j+1\}, p_j)$
    $c\{l+j+1\} \leftarrow c\{l+j+1\} \otimes [\hat{p}_j^{-1}]_{p_j}$
**end for**
**for** $i \leftarrow 0$ *to* $l$ **do**
    $c\{i\} \leftarrow \text{INTT}(\hat{c}\{i\}, q_i)$
    $s \leftarrow 0$
    **for** $j \leftarrow 0$ *to* $k-1$ **do**
        $s \leftarrow s + c\{l+j+1\} \otimes [\hat{p}_j]_{q_i} \bmod q_i$
    **end for**
    $c\{i\}' \leftarrow c\{i\} - s$
    $\hat{c}\{i\} \leftarrow \text{NTT}(c\{i\}', q_i)$
**end for**
Return $\hat{c} = (c\{0\}, ..., c\{l+k\})$

is done between $c\{i\}$ and a pre-calculated constant $\hat{q}_{i_{p_j}}$. Therefore, each coefficient of $c\{i\}$ needs to apply modulo $p_j$ before doing dyadic multiplication. The similar situation happens in Algorithm 7, where $c\{l+j+1\}$ is multiplied by $\hat{p_j}_{q_i}$. Therefore, there is a need to convert coefficients modular prime$_i$ to equivalent coefficient modular prime$_j$ under the congruent equivalent definition for ModUp and ModDown. When prime$_j$ is larger than prime$_i$, the coefficient is automatically pertained as it is. On the other side, when prime$_j$ is less than prime$_i$, we can usually make an easy modular switching by subtracting prime$_j$ from the coefficient modular$_i$. This is because the primes have a similar size due to CKKS's constraint on prime chain ($q_i/q_j \in (1-2^\eta, 1+2^\eta)$ for $i, j \in \{1, ..., L\}$). Only when prime$_i$ is $q_0$ and prime$_j$ is another prime number, we do a reduction by using the partially reduction-free technique because $q_0$ can be significantly larger than other primes. But it is guaranteed that $q_0$ is less than the square of any other number, so we can safely use the partially reduction-free technique for this modular switching.

# 4 Accelerator: CASA

## 4.1 Overall Structure

CASA is a compact and scalable accelerator designed by adopting the proposed design methodology. The top-level view of CASA is shown in Fig. 7. The key idea is to keep operations among different modules well-aligned so that an efficient parallel computation can be achieved. We use parameter $N_{bf}$ to denote the number of butterfly cores, but it is more crucial to the architecture layer because it also determines the number of dyadic multipliers and modulus switching units being used, as well as the bandwidth of memory I/O and data organization in the memory.

The challenging part is trying to keep as many modules working simultaneously as possible during a high-level operation. Since the NTT Module is the one that consumes the longest latency in the primitive function modules and needs to engage an entire Memory Bank, we set the delay for pipeline elements equal to the latency of an NTT/iNTT operation. Details of pipeline and parallel computation are introduced in Section 4.2.

CASA relies on off-chip memory to store the data that is not immediately used in parallel computation. The interface to connect with the off-chip memory is set on a Modulus Switching Bank to allow bit-width transition, i.e., the bit-width of off-chip

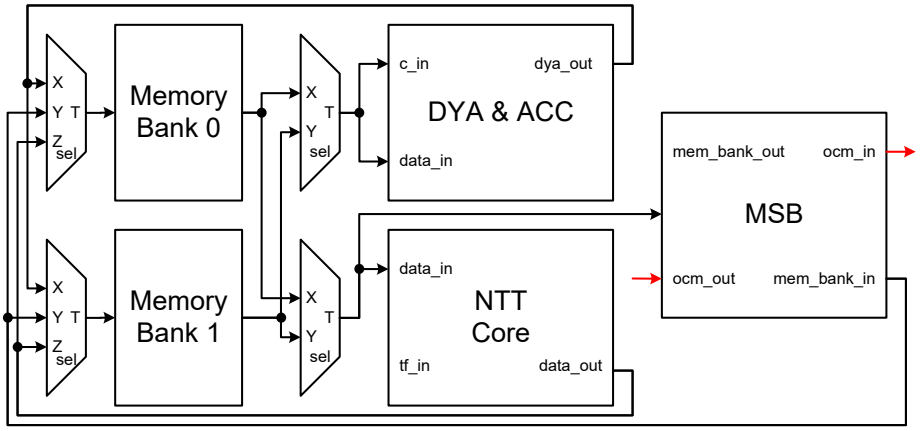



**Figure 7:** Top-level view of CASA. DYA & ACC: Dyadic and Accumulation. MSB: Modulus Switching Bank.

memory port usually comes at a power-of-two number such as 64, 128, and sometimes can go up to 1024 for AXI-4 interface on Artix-7.

The interface between CASA and external memory is set on Modulus Switching Bank because it is responsible for cutting out padded zeros and storing coefficients into on-chip memory (Memory Bank). Two 3-to-1 MUXes are placed before the two Memory Banks. This allows the NTT Module to occupy one Memory Bank while the other has access to MSB and DYA & ACC modules.

## 4.2 Scalable Parallel Computation

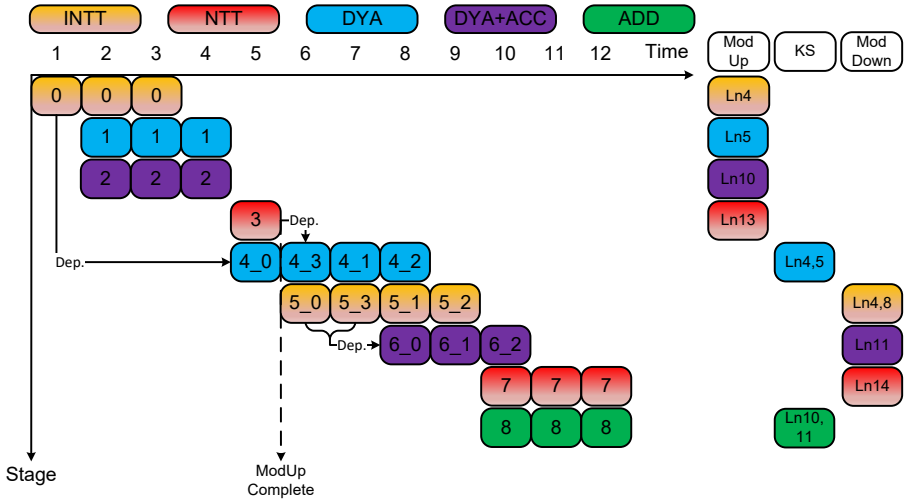



**Figure 8:** Key-Switching pipeline for $L = 3, K = 1$.

We propose a new hardware design methodology by first constructing the fine-tuned micro-architecture. However, the challenging part (making such a design work as a whole) is that the parallel computation needs to work smoothly and efficiently on top of those micro building blocks. In addition, to work on a range of devices as broadly as possible, we need to ensure the architecture has good scalability. Therefore, we design the parallel

computation strategy for CASA by the following steps. Firstly, we ensure the architecture is scalable straightforwardly by defining a single parameter for scaling up or down. The benefit is that we keep the entire structure scalable while any operation involved in the pipeline is implied. As introduced in Section 3.2, this design parameter is $N_{bf}$. Secondly, we consider the pipeline for key-switching (the most computationally-intensive operation in CKKS) as the bottleneck for efficient homomorphic circuit design. We study the pipeline of HEAX [RLPD20] and coxHE [HZL+22]. HEAX achieved a very high throughput by creating a cascaded structure containing multiple NTT Modules and potentially a different number of INTT modules. CoxHE proposed an optimized pipeline on top of HEAX and achieved shorter latency. However, both designs cost a tremendous hardware resource and rely on a pipeline that is challenging to scale. For example, HEAX requires {4 NTT Modules of 16 butterflies, 1 INTT Module of 16 butterflies, 2 INTT Modules of 8 butterflies, 3 Dyadic Modules, and 2 Modular Switching Modules} for $n = 2^{12}$. However, it is unclear how many modules of different types with different design parameters would be needed for $n = 2^{13}$. It turns out that {2 NTT Modules of 16 butterflies, and 2 Dyadic Modules} were added while the butterfly number of 2 INTT Modules was reduced from 8 to 4. [RLPD20] only disclosed limited information about the pipeline for different parameter sets, but this architectural change probably results in a significant change in its parallel computation. A re-design of parallel computation strategy may cost a huge amount of resource investment in real-world applications. CoxHE also involves a sophisticated architectural scaling process, but very limited information was reported. Therefore, we put the scalability of the proposed design as a priority when we design key-switching pipeline for CASA. To achieve this goal, we use the latency for NTT as the time for each element. The pipeline for $L = 3, K = 1$ is shown in Fig. 8. There are nine stages indexed from 0 to 8, and it takes 12 elements to finish the pipeline. The data dependency is denoted by Dep. No NTT or INTT operation is executed at the same time so that the pipeline is executable for one NTT Module. Since the bandwidth of the Memory Bank is organized to $N_{bf}$ coefficients (Fig. 6), the data flow goes into/out of the Memory Bank at each cycle is at the same width as the NTT Module, Dyadic and ACC Module, and Modulus Switching Bank. This square and aligned design allows the pipeline to work effectively, e.g., tightly connecting stage 1 to stage 2 (theoretically, stage 2 has to wait for stage 1 to fully accomplish due to the data dependency implied by ModUp Algorithm 6). But the data movement speed for a Memory Bank and a Dyadic and Acc Module is equal, so we can safely send the completed NTT data through the Dyadic and ACC Module before it goes to off-chip memory. From the perspective of the pipeline, stage 1 and stage 2 are finished at the same time. The same situation also happens in stages 7 and 8. For the key-switching algorithm, the CKKS ciphertext contains two parts $(c_0, c_1)$, so we implement two identical structures as shown in Fig. 7 to process the key-switching pipeline. Another important aspect of acquiring highly-efficient parallel computation hardware is to keep as much component work at the same time as possible. Since the in-place NTT Module engages an entire Memory Bank while working, the goal is that other modules can keep working. We add two MUXes for the two Memory Banks so that all Memory banks can keep working with a computational module or exchange data with off-chip memory.

## 5 Implementation & Comparison

We have coded our accelerator with VHDL (with functionality verified) and implemented by Vivado 2020.1 on the AMD-Xilinx ZCU-102 FPGA evaluation board. The implemented designs are categorized by different CKKS parameters and the design parameter $N_{bf}$. We have followed the CKKS parameter sets from [RLPD20] to implement them so that the security and accuracy are directly inherited. Note that the design parameter $N_{bf}$ is the number of butterfly cores within an NTT Module, which also implies the number of ports

**Table 1:** CKKS Parameter Sets (From [RLPD20])

| Parameter Set | $n$ | $\log_P Q$ | $L$+1 | Homo. Op. Level | Sec. Level/bit |
|---|---|---|---|---|---|
| I | $2^{12}$ | 109 | 2 | 1 | 128 |
| II | $2^{13}$ | 218 | 4 | 3 | 128 |
| III | $2^{14}$ | 438 | 8 | 7 | 128 |

in the Memory Bank, Partially Reduction-Free Multipliers in the Modulus Switching Bank, and Dyadic Multiplication Module (as described in Section 4).

The security configurations are directly borrowed from [ACC+21] to ensure at least 128-bit security against computational attack. To make a straightforward comparison with the existing reports in the literature, we select three parameter sets that can be uniquely identified by $n$ for $n = 2^{12}, 2^{13}$ and $2^{14}$ as shown in Table 1. The number of primes in the prime-chain for $p_i$ is $L + 1$ and the homomorphic operation level is $L$.

**Table 2:** Area Usage Comparison with The Existing Hardware CKKS Accelerators (And an FV Design)

| Design | Device | $n$ | LUT | FF | DSP | BRAM | Extra Memory (OFCM,URAM) |
|---|---|---|---|---|---|---|---|
| [HZL+22] | ZCU102 | 8,192 | 153k | 115k | 2420 | 639 | (Y,Y) |
| CASA ($N_{bf}$ =2) | ZCU102 | 8,192 | 19k | 3k | 108 | 52 | (Y,N) |
| CASA ($N_{bf}$ =4) | ZCU102 | 8,192 | 33k | 5k | 216 | 56 | (Y,N) |
| [RLPD20] | Stratix10 | 16,384 | 1,199k | 1,746k | 2,370 | 5,183 | (Y,N) |
| [MKS+22] | Alveo U250 | 16,384 | 963k | 669k | 3,600 | 1,280 | (N,Y) |
| CASA ($N_{bf}$ =4) | ZCU102 | 16,384 | 33k | 6k | 216 | 104 | (Y,N) |
| CASA ($N_{bf}$ =8) | ZCU102 | 16,384 | 66k | 11k | 432 | 112 | (Y,N) |
| [RTJ+19][1] | ZCU102 | 16,384 | 63,522 | 25,622 | 815 | 208 | (N,N) |

OFCM: off-chip memory. URAM: ultra RAM.
[1]: The targeted FHE scheme is FV (the other works are all designed for CKKS).

We report the area usage of the entire design and the key-switching timing results in Tables 2 and 3, respectively. For parameter set I, we report two implemented areas for $N_{bf} = 2$ and 4; while for parameter set II, we reported the results for the cases of $N_{bf} = 4$ and 8. Under these settings, the proposed CASA has shown superior overall area-time performance compared to the existing accelerator based on the normalized metric of area-delay product (ADP). Note that to conduct a fair comparison between different designs implemented on different devices, we follow [LFK+19, THKX23] to assign different equivalent ratios to specific hardware resources (LUT, FF, DSP, and BRAM) to calculate the overall area usage, as mentioned in Table 3. We also report the implementation results on a resource-constrained Artix-7 device. We choose $N_{bf} = 8$ and $n = 2^{13}$ and $n = 2^{14}$ to demonstrate the compactness of proposed design.

**Comparison to HEAX [RLPD20].** HEAX is the first full-hardware implementation for CKKS that includes homomorphic multiplication, homomorphic addition, and key-switching. It embraces a dedicated cascaded structure to enable a highly paralleled pipeline dataflow and demonstrated a much higher processing speed, compared to previous software-based implementations including CPU and GPU. Even though HEAX can generate a much higher throughput with a significant area consumption (over 24x compared to CASA of $N_{bf} = 4$) to maintain the high processing performance, it suffers from side-effects of the sophiscated cascaded hardware structure. Compared with HEAX, CASA has at least the following advantages. (i) Lower delay measured by per-unit of area cost. We argue that constructing a high-throughput processor in one single design with complex structure has less practical usage value than focusing on the execution delay per operation. (ii) Simplicity in terms of real-world implementations. Common industrial demands for a

**Table 3:** Comprehensive Performance Comparison for Key-Switching Operation

| Design | Device | $N$ | Area Overall[1] | Fmax | Latency | Delay/$\mu s$ | ADP |
|---|---|---|---|---|---|---|---|
| [HZL+22] | ZCU102 | 8,192 | 309,972 | - | - | 593.84* | 184.07 |
| CASA ($N_{bf}$ =2) | ZCU102 | 8,192 | 16,624 | 185 | 319488 | 3,463 | 57.57 |
| CASA ($N_{bf}$ =4) | ZCU102 | 8,192 | 29,698 | 185 | 159,744 | 1,725 | 51.23 |
| [RLPD20] | Stratix10 | 16,384 | 792,013 | 300 | - | 1,195 | 946.19 |
| [MKS+22] | Alveo U250 | 16,384 | 602,546 | 200 | 95,352 | 476.76 | 287.27 |
| CASA ($N_{bf}$ =4) | ZCU102 | 16,384 | 32,434 | 175 | 688,128 | 7,845 | 254.43 |
| CASA ($N_{bf}$ =8) | ZCU102 | 16,384 | 59,450 | 174 | 344,064 | 3,964 | 235.63 |
| [RTJ+19][2] | ZCU102 | 16,384 | 104,645 | 200 | - | 29.3k[3] | 3,062 |

[1]: We follow [LFK+19, THKX23] to assign different equivalent ratios to specific hardware resources and calculated area overall as LUT/16+FF/8+DSP×102.4+BRAM×56.
[2]: The targeted FHE scheme is FV (rather than CKKS).
[3]: Delay is reported for homomorphic multiplication.
*: The design of [HZL+22] did not report the maximum frequency, we assume this design can run at 250MHZ (the maximum frequency of the evaluation board) to estimate the delay.

high-performance design with robustness and simplicity. From a practical perspective, a simple structure also implies a larger optimization space and lower manufacturing cost per unit. (iii) Overall better area-time complexities, at least 3.7x ADP over CASA ($N_{bf} = 4$).

**Comparison to coxHE [HZL+22].** Accelerator coxHE is a successor of HEAX [RLPD20], which optimized the design parameters and pipeline strategy. The major contribution of this work was a novel processing procedure. It claims 33-48% reduced normalized key-switching latency compared with HEAX [HZL+22]. However, the detailed timing and area information, such as the latency of key-switching operation, maximum frequency of the processor, were not reported for making a straightforward comparison with HEAX. Due to limited information disclosed by coxHE, we prepare a careful evaluation based on the key-switching algorithm proposed by the authors to estimate the key-switching latency. Firstly, we pick parameter set II, which is the largest parameter reported by coxHE, and we acquire the best amongst three listed designs in Table 3 (original paper) in terms of overall performance. Only a bar chart in the original paper (Fig. 5.(b) of [RLPD20]) was given to indicate the key-switching cycle, i.e., when the number of cycles was approximately 200,000 for HEAX for 4 moduli, coxHE claims 11.4%-25.77% overall performance enhancement. In the best scenario for coxHE, we estimated the number of cycles as $200,000 \times (1 - 25.77\%)$. Meanwhile, due to the absence of the frequency report, we estimate the maximum frequency allowed by coxHE at 250MHz, which is the maximum allowed on-chip frequency of the target evaluation board. Under this best setting for coxHE, CASA still achieves 3.2-3.6x overall performance measured by ADP, which is due to the significant reduction in area usage. Another factor that could potentially contribute to this comparison is that coxHE is implemented in HLS while CASA is implemented by VHDL which is more suitable for describing micro-architectures.

**Comparison to Medha [MKS+22].** Medha proposes a high-performance ring-like architecture for CKKS implementation. Medha was evaluated on an advanced FPGA board Alevo U250, which is designed for data center and cloud service usage. It runs at 200 MHz with 10 Residue Polynomial Arithmetic Unit (RAPUs). When all RAPUs were carefully placed to form a ring for smooth data flow, Medha achieved comparable throughput compared to HEAX [RLPD20] while it was 2x faster in terms of latency in multiplication + key-switching. Since 95.9% of the time in executing (multiplication + key-switching) operations on Mehda was on key-switching, we mainly compare its key-switching performance with CASA. At parameter set III, Mehda achieves approximately 16.5x and 8.3x speedup compared to CASA when $N_{bf}$ is configured to 4 and 8, respectively. This comes at a cost of 18.6x and 10.1x overall area consumption over CASA, and it turns out to be that CASA achieves 12.9% and 21.9% performance enhancement in ADP

over Medha. We want to highlight that not only the slight increment in area-timing performance, but also two come along factors proving CASA's architectural advantages. (i) CASA does not require a ring-like placement of a certain number of RPAUs, so it demonstrates more scalability in practical usage. The need for routing data flow among different RPAUs also implies dramatically increased placing and routing challenges for hardware compilers. While Medha was only reported for evaluation on the data center FPGA, CASA also reports the results on a resource-constrained device. (ii) The intended simplicity in high-level architecture and control allows a quicker development and larger optimization space. Data movement between major components is squared and aligned to design parameter $N_{bf}$, which reduces the possibility of significantly modifying the data bus structure inside the accelerator. When it comes to the design of control systems, the way of taking pipeline strategy for parallel computation is also arguably easier in programming and more extendable compared to controlling Medha's ring-like RPAUs.

Comparison to [RTJ+19]. [RTJ+19] presented an FV accelerator based on the FPGA platform. We used homomorphic multiplication time to evaluate the timing efficiency for [RTJ+19] because its actual latency is not reported. With the same polynomial size, CASA achieves about 7.4x faster speed than [RTJ+19]. Meanwhile, CASA consumes considerably less resource usage than [RTJ+19]. Overall, the timing efficiency (of CASA) and much smaller area consumption result in a 13.0x reduction in ADP (see Table 3).

**Table 4:** Complexities for CASA on Artix-7 Device

| Design/Module | LUT | FF | DSP | BRAM | Fmax/MHz |
|---|---|---|---|---|---|
| Param. Set-II, $N_{bf} = 8$ | | | | | |
| CASA | 65,294 (48.51%) | 11,058 (4.11%) | 432 (58.38%) | 40 (10.96%) | 88 |
| NTT | 21,059 (15.65%) | 1,037 (0.39%) | 0 (0.00%) | 0 (0.00%) | |
| DYA & ACC | 32,656 (24.26%) | 932 (0.35%) | 0 (0.00%) | 0 (0.00%) | |
| MSB | 8,086 (6.01%) | 9,089 (3.38%) | 432 (58.38%) | 0 (0.00%) | |
| MemBank | 0 (0.00%) | 0 (0.00%) | 0 (0.00%) | 40 (10.96%) | |
| Ctrl & Interface | 3,493 (2.60%) | 0 (0.00%) | 0 (0.00%) | 0 (0.00%) | |
| Param. Set-III, $N_{bf} = 8$ | | | | | |
| CASA | 67,587 (50.22%) | 11,094 (4.12%) | 432 (58.38%) | 56 (15.34%) | 85 |
| NTT | 23,618 (17.55%) | 1,021 (0.38%) | 0 (0.00%) | 0 (0.00%) | |
| DYA & ACC | 32,190 (23.92%) | 1,033(0.38%) | 0 (0.00%) | 0 (0.00%) | |
| MSB | 8,126 (6.04%) | 9,040 (3.36%) | 432 (58.38%) | 0 (0.00%) | |
| MemBank | 251 (0.19%) | 0 (0.00%) | 0 (0.00%) | 56 (15.34%) | |
| Ctrl & Interface | 3,402 (2.53%) | 0 (0.00%) | 0 (0.00%) | 0 (0.00%) | |

This Artix-7 device (xc7a200tffg1156-3) has 134,600 LUTs, 269,200 FFs, 740 DSPs, and 365 BRAMs.

**Discussion about the Implementation on Artix-7 Device.** The results in Table 4 fully demonstrate the compactness of the proposed CASA (as well as its potential in resource-constrained applications), i.e., with resource usage of around 49% LUTs, 4.1% FFs, 58% DSPs, and 11%/15% BRAMs, CASA still maintains an excellent timing.

**Comparison to CPU [Mic20] and GPU [ABHM+20] implementations.** Table 5 shows the performance comparison with CPU and GPU implementations. [Mic20] is the SEAL library developed by Microsoft. SEAL is implemented in C++ and employs many highly-optimized programming techniques, which is also fine-tuned on multi-thread operations. SEAL takes 418 and 33,840 $\mu s$ to execute one homomorphic addition, multiplication, and rescaling for parameter set II, respectively. CASA achieves 10.84x speedup while only consuming 5.8% power (compared with [RLPD20]). Note the GPU acceleration of [ABHM+20] is the only work implementing CKKS for the given parameter set, though $\log_2 Q$ is slightly different (360-bit for [ABHM+20]). In this work [ABHM+20], the authors used NVIDIA DGX-1 GPU (consisting of 8 NVIDIA V100 cores) to execute CKKS for parameter set III. In terms of sheer speed, [ABHM+20] is 4.8x faster in homomorphic

**Table 5:** Performance Comparison with Recent CPU & GPU Implementations

| Design | Platform | Homo. Add./$\mu s$ | Homo. Mult./$\mu s$ | Resc. /$\mu s$ | Power /W | PDP[3] Add. | PDP[3] Mult. | PDP[3] Resc. |
|---|---|---|---|---|---|---|---|---|
| Param. Set-II | | | | | | | | |
| [RLPD20] | CPU[1] | - | 11,905 | - | 85.0 | - | 1,012 | - |
| CASA ($N_{bf}$ =8) | Artix-7 | 47 | 1,097 | 379 | 5.0 | 0.23 | 5.43 | 1.9 |
| Param. Set-III | | | | | | | | |
| [ABHM+20] | GPU[2] | 40 | 740 | 0.14 | 3,500 | 140 | 2,590 | 490 |
| CASA ($N_{bf}$ =8) | Artix-7 | 193 | 4,833 | 1,522 | 5.1 | 1.0 | 24.7 | 7.8 |

Add.: homomorphic addition; Mult.: homomorphic multiplication; Resc.: rescaling.
[1] CPU used is a single-threaded Intel Xeon(R) Silver 4108 @ 1.8GHz.
[2] GPU used is NVIDIA DGX-1 (8 NVIDIA V100 cores).
[3] Power-timing product (PTP): power consumption (Walt) multiplied by operational latency (millisecond).

**Table 6:** Comparison of Throughput with Software and Hardware Implementations

| Design | Scheme | Platform | Thru./s | Thru./Area[1] | Thru./Watt |
|---|---|---|---|---|---|
| [Mic20] | RNS-CKKS | CPU | 29.55 | - | 0.49 |
| [ABHM+20] | RNS-CKKS | GPU | 1,351 | - | 0.39 |
| [RLPD20] | RNS-CKKS | Stratix10 | 2,616 | 3.30 | - |
| [MKS+22] | RNS-CKKS | Alevo U250 | 2,011 | 3.33 | 32.37 |
| CASA ($N_{bf}$ =8) | RNS-CKKS | ZCU102 | 423.9 | 7.13 | 89.76 |
| CASA ($N_{bf}$ =8) | RNS-CKKS | Artix-7 | 206.89 | 3.66 | 40.52 |
| [RTJ+19] | FV | ZCU102 | 33.78 | 0.32 | - |

Polynomial degree is $2^{14}$ for each RNS-CKKS and FV design.
1. Throughput per area unit is throughput per second divided by area. Area calculation is the same as introduced in Table. 3. The presented value is multiplied by 1,000 for clarity.

addition, 6.5x faster in homomorphic multiplication, and 10.9x faster in rescaling. However, if power consumption is taken into account, the peak power consumption of [ABHM+20] is about 700x higher than CASA, which results in an approximately 141x, 104x, and 63x advantage for CASA in terms of computation per Watt.

**Throughput Evaluation.** While we emphasize CASA's flexibility and compatibility by showing it does not apply to specific devices, especially devices of high power or manufacturing cost, we compare CASA's throughput (when $N_{bf}$ =8) with larger accelerator designs. As shown in Table 6, CASA's high-level homomorphic operation (multiplication + key-switching) throughput is listed along with CPU, GPU, and those recent FHE FPGA accelerators. Due to the resource usage and design style of different implementations, we use two metrics, namely Throughput/Area and Throughput/Watt, to evaluate a design's overall throughput efficiency.

We evaluate CASA's throughput performance on an advanced ZCU102 device (still less advanced than Alveo) and a mainstream resource-constrained Artix-7 device. Considering the performance on the ZCU102 device, though CASA's throughput is 10.8% per second compared to HEAX [RLPD20], CASA achieves 1.48 times better efficiency than [RLPD20] when the area utilization is taken into account (i.e., Throughput/Area). Mehda's [MKS+22] sheer throughput is less than HEAX [RLPD20], but it achieves better performance when throughput is normalized by area consumption (Throughput/Area). Since Mehda [MKS+22] also reported power usage, we evaluate power normalized throughput against Mehda [MKS+22], i.e., Throughput/Watt (also can be called energy efficiency). Even though CASA uses ZCU102 (smaller than Medha used Alveo [MKS+22]), CASA achieves 2.77 times better Throughput/Watt. On the Artix-7 device, CASA is 12.64 and 9.72 times slower compared to [RLPD20] and [MKS+22] from a sheer amount perspective

(we want to note that these two designs come with much more advanced devices and much larger manufacturing costs). If we take power consumption into account, CASA achieves 79.9% Throughput/Watt to Mehda [MKS+22]. Meanwhile, CASA obtains 1.11x and 1.10x Throughput/Area compared to HEAX [RLPD20] and Mehda [MKS+22], respectively (taking the devices' differences and manufacturing costs into account, our Artix-7 CASA actually involves much more efficiency than HEAX [RLPD20] and Mehda [MKS+22]). Finally, compared with FV implementation [RTJ+19], CASA achieves 6.12 times enhancement on throughput and consumes a much smaller area.

When comparing with GPU/CPU implementation of [ABHM+20] and [Mic20], we want to emphasize that CASA (on the Artix-7 device) has 105x Throughput/Watt than [ABHM+20] and 7.0 times higher throughput (sheer volume) than [Mic20] (while consuming 1/12 power). CASA on the ZCU102 device has much better throughput efficiency than the CPU and GPU implementations of [ABHM+20] and [Mic20].

**Application Discussion.** There are several real-word applications where low-degree FHE, homomorphic evaluation, power efficiency, and low manufacturing costs are preferred. One interesting application scenario is intelligent traffic navigation, which requires the anonymity of vehicle users, while navigation needs to be real-time and energy efficient. Very recently, [ZCC+23] proposed an efficient cryptographic protocol using multi-party FHE key in road side unit (RSU) to achieve multiple benefits in a large scaled vehicular network. The hardware part of the solution needs to place RSU on the critical points of each road to communicate with vehicles and cloud service providers (CSP). To balance anonymity and efficiency, RSU needs to return to CSP the average velocity and position of vehicles on that road. It requires RSU to do a low-degree homomorphic evaluation in real-time and before sending it back to CSP. Since each road will need a budget for installing RSU, small devices would be preferred, and thus, the proposed CASA would be more desirable than those much larger accelerators. Other real-world examples would be collaboration over a multi-party shared project or dataset where one or several participating parties possess only resource-constraint devices but must apply critical operations over encrypted messages. A possible scenario can be, for instance, the collaborated work needs to be signed by one party who has only a wearable device. As Aritx-7 device has already been deployed in the IoT environment [MMM+21, IoT20], we are expecting that CASA can be used for many similar emerging applications.

**Scalability Consideration**. As CASA is a compact design with scalability, we demonstrate its efficiency in these two aspects when we choose $N_{bf}$ from 2, 4, to 8 (Tables 2, 3, and 4). As shown in Table 3, CASA can be easily scaled for $N = 2^{13}$ and $N = 2^{14}$ by altering the design parameter $N_{bf}$. CASA's overall area usage is closely related to $N_{bf}$. When $N_{bf}$ is doubled, all kinds of hardware resource usage also almost doubled, i.e., CASA's structure is well-aligned with design parameter $N_{bf}$. Meanwhile, CASA's operational frequency is relatively stable even the design parameter $N_{bf}$ changes. Overall, the proposed CASA follows a pattern that its resource usage will be doubled again if we choose $N_{bf} = 16$, making CASA lose its compactness and applicability in low-end FPGA devices. Hence, we chose the scale parameter $N_{bf}$ from 2 to 8. With this type of scalability setup, CASA is able to be deployed in a large variety of application scenarios, covering both high-end and low-end applications (especially desirable for resource-constrained applications), depending on the selection of the scale parameter and related FPGA devices.

**Discussion.** It is clear that CASA obtains better area-timing performance than the existing designs while possessing excellent scalability. Meanwhile, CASA also demonstrated its high potentiality in resource-constrained applications (first report in the literature).

Note that the practical setting for FHE applications usually can be divided into two categories: (I) operations over ciphertext; and (II) interactions with plaintext and keys including key-generation, encoding, and decoding. In order to focus on accelerating the practical application of CKKS, we omitted category II in CASA since these operations

are not carried-out in batch but require a significant hardware resource (the same in the existing designs like [RLPD20, HZL+22, MKS+22]). It is also likely that the operations in this category are more economic if to be left as a CPU/GPU job.

**Other FHE Acceleration Works.** Other hardware implementations for FHE accelerators include FV implementations [RJV+18, RTJ+19, TRV20], BFV implementation [SYYZ22]. As these designs targeted different FHE schemes, a direction comparison between CASA and these designs is not thus feasible (following the discussion in [MKS+22]). Nevertheless, as shown in Tables 2, 3, and 6, the proposed CASA has demonstrated its efficiency over the existing FV accelerator of [RTJ+19].

Meanwhile, other works [SFK+21, SFK+22, KKK+22, KLK+22] have proposed different ASIC architectures and simulations. However, as stated in Section 1.1, these ASIC designs suffer from several bottleneck drawbacks (following [MMM+21]), and their performance results were obtained through estimations. [NSA+22] reported an accelerator for CKKS based on a smaller parameter set but fabricated a real ASIC chip. Overall, we don't explicitly compare them here as they were already extensively discussed in previous works such as [MKS+22].

**Future Works and Direction.** While the proposed CASA is highly efficient in terms of compactness and scalability, more arithmetic innovations are expected to be carried out due to the huge computational complexity of CKKS (and other FHE schemes). Breakthroughs in the accelerator's design methodology need serious investigation as building such a large-scale accelerator is not a trivial effort.

# 6 Conclusions

In this paper, we present a compact and scalable accelerator suitable for the practical use of homomorphic technique. Several innovative techniques are applied to obtain a compact design target, including novel partially reduction-free modular arithmetic, modular switching bank multiplexing, and dataflow optimization. The proposed CASA is implemented on a resource-constrained FPGA resulting compact area usage and excellent timing, which is the first report in the literature. Compared with the state-of-the-art designs, CASA obtains significantly better overall performance (including the Artix-7 implemented CASA).

# Acknowledgement

Ç. K. Koç was supported by TUBITAK Project 1001-121F348. J. Xie was supported in part by NIST-60NANB20D203 and NSF SaTC-2020625.

# References

[ABBB+22] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.

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

[ACC+21] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter,

et al. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, pages 31–62, 2021.

[AS10] Tolga Acar and Dan Shumow. Modular reduction without pre-computation for special moduli. *Microsoft Research*, 2010.

[Bar00] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology—CRYPTO'86: Proceedings*, pages 311–323. Springer, 2000.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[Bla83] G. R. Blakley. A computer algorithm for the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 868–886. Springer, 2012.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 3–33. Springer, 2016.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.

[DIZ07] Vassil Dimitrov, Laurent Imbert, and Andrew Zakaluzny. Multiplication by a constant is sublinear. In *18th IEEE Symposium on Computer Arithmetic (ARITH'07)*, pages 261–268. IEEE, 2007.

[DNKYL22] Phap Duong-Ngoc, Sunmin Kwon, Donghoon Yoo, and Hanho Lee. Area-efficient number theoretic transform architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[Gen09a] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[HZL+22] Mingqin Han, Yilan Zhu, Qian Lou, Zimeng Zhou, Shanqing Guo, and Lei Ju. coxHE: A software-hardware co-design framework for FPGA acceleration of homomorphic computation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1353–1358. IEEE, 2022.

[IoT20] Remote monitoring and control of home appliances from cloud using EDGE Artix 7 FPGA board, https://allaboutfpga.com/remote-monitoring-and-control-of-home-appliances-from-cloud-using-edge-artix-7-fpga-board/, 2020.

[KA98] Ç. K. Koç and T. Acar. Montgomery multiplication in GF($2^k$). 14(1):57–69, April 1998.

[KAK96] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 711–725, 2022.

[KLC+20] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A Rutenbar. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 56–64. IEEE, 2020.

[KLK+22] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1237–1254. IEEE, 2022.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers by automata. *Soviet Physics-Doklady*, 7:595–596, 1963.

[KT05] M. E. Kaihara and N. Takagi. Bipartite Modular Multiplication Method. *Cryptographic Hardware and Embedded Systems - CHES 2005*, pages 201–210, 2005.

[KT08] Marcelo Kaihara and Naofumi Takagi. Bipartite modular multiplication method. *IEEE Transactions on Computers*, 57(2):157–164, 2008.

[LFK+19] Weiqiang Liu, Sailong Fan, Ayesha Khalid, Ciara Rafferty, and Máire O'Neill. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(10):2459–2463, 2019.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15*, pages 124–139. Springer, 2016.

[Mic20] Microsoft. SEAL release. *https://github.com/Microsoft/SEAL*, 2020.

[MKS+22] Ahmet Can Mert, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, Sujoy Sinha Roy, et al. Medha: Microcoded hardware accelerator for computing on encrypted data. *arXiv preprint arXiv:2210.05476*, 2022.

[MMM+21] Mahabub Hasan Mahalat, Suraj Mandal, Anindan Mondal, Bibhash Sen, and Rajat Subhra Chakraborty. Implementation, characterization and application of path changing switch based arbiter PUF on FPGA as a lightweight security primitive for iot. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(3):1–26, 2021.

[Mon85] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[NSA+22] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. CoFHEE: A co-processor for fully homomorphic encryption execution. *arXiv preprint arXiv:2204.08742*, 2022.

[OMSL+23] Samira Carolina Oliva Madrigal, Gökay Saldamlı, Chen Li, Yue Geng, Jing Tian, Zhongfeng Wang, and Çetin Kaya Koç. Reduction-free multiplication for finite fields and polynomial rings. In Sihem Mesnager and Zhengchun Zhou, editors, *Arithmetic of Finite Fields*, pages 53–78, Cham, 2023. Springer International Publishing.

[PS21] Rogério Paludo and Leonel Sousa. Number theoretic transform architecture suitable to lattice-based fully-homomorphic encryption. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 163–170. IEEE, 2021.

[RJV+18] 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.

[RLPD20] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1295–1309, 2020.

[RTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*, pages 387–398. IEEE, 2019.

[SBE15] Gokay Saldamli, YoJin Baek, and Levent Ertaul. Partially Interleaved Modular Karatsuba-Ofman Multiplication. *IJCSNS*, 15(5):503–518, may 2015.

[SCH71] A. SCHONHAGE. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[SFK+21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021.

[SFK+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: a hardware accelerator for efficient unbounded

computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 173–187, 2022.

[SYYZ22] Yang Su, Bai-Long Yang, Chen Yang, and Song-Yin Zhao. ReMCA: A reconfigurable multi-core architecture for full RNS variant of BFV homomorphic evaluation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2857–2870, 2022.

[THKX23] Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and Jiafeng Xie. LEAP: Lightweight and efficient accelerator for sparse polynomial multiplication of HQC. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

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

[ZCC+23] Jun Zhou, Shiying Chen, Kim-Kwang Raymond Choo, Zhenfu Cao, and Xiaolei Dong. EPNS: Efficient privacy-preserving intelligent traffic navigation from multiparty delegated computation in cloud-assisted vanets. *IEEE Transactions on Mobile Computing*, 22(3):1491–1506, 2023.

[ZLL+21] Cong Zhang, Dongsheng Liu, Xingjie Liu, Xuecheng Zou, Guangda Niu, Bo Liu, and Quming Jiang. Towards efficient hardware implementation of NTT for Kyber on FPGAs. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.