Aloha-HE: A Low-Area Hardware Accelerator for Client-Side Operations in Homomorphic Encryption

Florian Krieger, Florian Hirner, Ahmet Can Mert, Sujoy Sinha Roy
Institute of Applied Information Processing and Communications, Graz University of Technology, Graz, Austria
{florian.krieger, florian.hirner, ahmet.mert, sujoy.sinharoy}@iaik.tugraz.at

***Abstract*—Homomorphic encryption (HE) has gained broad attention in recent years as it allows computations on encrypted data enabling secure cloud computing. Deploying HE presents a notable challenge since it introduces a performance overhead by orders of magnitude. Hence, most works target accelerating server-side operations on hardware platforms, while little attention has been given to client-side operations. In this paper, we present a novel design methodology to implement and accelerate the client-side HE operations on area-constrained hardware. We show how to design an optimized floating-point unit tailored for the encoding of complex values. In addition, we introduce a novel hardware-friendly algorithm for modulo-reduction of floating-point numbers and propose various concepts for achieving efficient resource sharing between modular ring and floating-point arithmetic. Finally, we use this methodology to implement an end-to-end hardware accelerator, Aloha-HE, for the client-side operations of the CKKS scheme. In contrast to existing work, Aloha-HE supports both encoding and encryption and their counterparts within a unified architecture. Aloha-HE achieves a speedup of up to 59× compared to prior hardware solutions.**

***Index Terms*—CKKS, Homomorphic Encryption, Hardware Accelerator, FPGA, Microsoft SEAL.**

I. Introduction

Homomorphic Encryption (HE) is a special technique that allows direct computation on the encrypted data, meaning that operations on the encrypted data yield the same result as on its plain counterpart. This property allows to preserve the confidentiality of data in the context of cloud computing. In the HE setting, there are two parties, client and cloud. A client encodes and encrypts its data, and sends it to the cloud in encrypted format. The cloud performs computations homomorphically on the encrypted data while keeping it confidential. Finally, the client reads the processed data from the cloud in the encrypted format, and then decrypts and decodes it. There are different HE schemes in the literature [1], [2].

The CKKS scheme [2] has emerged as a promising HE scheme as it allows computations on real numbers, hence enabling many applications such as privacy-preserving neural networks. Computations on real numbers add further complexity to the CKKS scheme since it requires two distinct transformations. First, the Fast Fourier transform (FFT), which operates on complex numbers, is used during encoding. It maps the complex-valued input vector to a polynomial ring element before its encryption. The second transformation used in CKKS is the Number Theoretic Transform (NTT). It operates on polynomial ring elements and ensures performant polynomial multiplications during homomorphic computations.

Despite HE enables secure computation, its adoption still suffers from its massive memory and computational requirements, which are several orders of magnitude higher compared to plaintext operations. Recently, the United States Department of Defense initiated the DPRIVE competition [3] to boost the development of HE schemes, including acceleration of HE on hardware platforms. Most research works as well as the DPRIVE competition focus on accelerating cloud-side operations on high-end FPGA and ASIC platforms [4], [5]. Yet, little attention is given to client-side operations including encryption, encoding, and their respective counterparts. In many cases, a low-end device is deployed on the client side. This makes it challenging to perform required operations while meeting various resource and performance constraints. These challenges inspired recent works to design solutions targeting client-side operations on either software or hardware platforms. SEAL-Embedded [6] is a software library optimized for low memory consumption to allow its use on embedded devices for CKKS. However, it comes at a cost of high latency. There are also works targeting hardware acceleration for client-side CKKS operations [7]–[9]. They provide support for encryption and its sub-operations including NTT. The major drawback of these designs is the expensive CKKS encoding performed in software as it requires floating-point computations during FFT.

This paper presents Aloha-HE, an end-to-end hardware accelerator for client-side CKKS operations. Aloha-HE is the first work that provides support for encoding, encryption, and their counterparts in hardware while prior hardware works only provide support for encryption. The main contributions of this work can be summarized in five points. (1) We implement an FFT unit including an IEEE-754 compliant floating-point unit. Our design benefits from a high degree of resource sharing between FFT and NTT units. (2) We introduce a novel and hardware-friendly approach to compute the modular reduction of floating-point values by exploiting the IEEE-754 format. (3) We implemented a co-processor for accelerating client-side HE operations. As a demonstration, we target CKKS scheme. (4) Our implementation, Aloha-HE, on low-cost FPGAs shows a speedup of up to 59× compared to prior works. (5) We publish our source code at [10] to support further research in this area.

This paper is structured as follows. Sec. II gives notation and background for CKKS scheme. Sec. III presents our accelerator design. Sec. IV shows our results and Sec. V concludes paper.

This project was funded in part by the CONFIDENTIAL-6G EU project (Grant No: 101096435). It was also supported in part by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.

979-8-3503-4859-0/DATE24/© 2024 EDAA

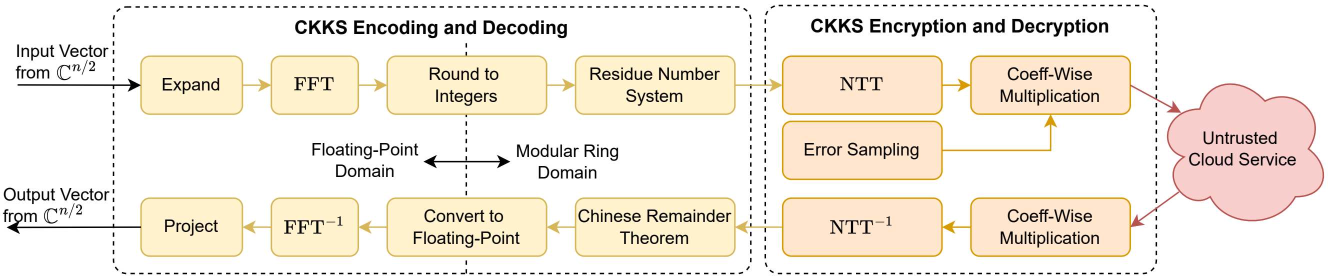
CKKS Encoding and Decoding
CKKS Encryption and Decryption
Input Vector from $\mathbb{C}^{n/2}$
Expand
FFT
Round to Integers
Residue Number System
NTT
Coeff-Wise Multiplication
Error Sampling
Untrusted Cloud Service
Floating-Point Domain
Modular Ring Domain
Output Vector from $\mathbb{C}^{n/2}$
Project
FFT^{-1}
Convert to Floating-Point
Chinese Remainder Theorem
NTT^{-1}
Coeff-Wise Multiplication

Fig. 1. Flow chart of the CKKS encoding+encryption and decoding+decryption operations.

II. Background

In this section, we present notation with required mathematical building blocks and introduce the CKKS scheme.

A. Notation

Lattice-based cryptosystems relying on the Ring Learning-With-Errors (RLWE) commonly operate over a polynomial ring denoted by $\mathbf{R}_Q = \mathbb{Z}_Q[x]/(x^n + 1)$. This is the set of polynomials with degree at most $n - 1$ and with coefficients from $\mathbb{Z}_Q$. The $2n$-th cyclotomic polynomial $\Phi_{2n}(x) = x^n + 1$ serves as the irreducible polynomial required for reducing a polynomial multiplication result back to $\mathbf{R}_Q$. We refer to Q as ciphertext modulus, which is a product of distinct primes $Q = q_0 \cdot \ldots \cdot q_{L-1}$ with a maximum bit-size defined by the polynomial size n to ensure a certain security level [11]. Each prime q_i has the property $q_i \equiv 1 \mod 2n$ and thus enables Number Theoretic Transform (NTT) based polynomial multiplication in $\mathbf{R}_Q$. Throughout this paper, we refer to integer and polynomial multiplication as $\cdot$ and $\times$, respectively.

B. Mathematical Building Blocks

The Fast Fourier Transform (FFT) is an efficient approach to compute the Discrete Fourier Transform (DFT) of a complex-valued input vector $\mathbf{x} \in \mathbb{C}^n$. It iterates a so-called butterfly operation $\frac{n}{2}\log_2(n)$ times over the input vector $\mathbf{x}$. There exist two types of butterfly configurations, the Gentleman-Sande (GS) butterfly is used in decimation-in-frequency (DIF) transformations while the Cooley-Tukey (CT) butterfly is used in decimation-in-time (DIT) transformations [12]. Each butterfly operation gets two coefficients as input and produces two as output. Additionally, a $2n$-th primitive root of unity, the so-called twiddle factor ω, is used during butterfly computations [4]. As the FFT is defined over the field of complex numbers $\mathbb{C}$, floating-point computations are required in digital systems. Each complex value $x = x_r + jx_i \in \mathbb{C}$ consists of two floating-point values, the real part $x_r \in \mathbb{R}$ and the imaginary part $x_i \in \mathbb{R}$. Complex addition $a + b = (a_r + b_r) + j(a_i + b_i)$ requires two floating-point additions, while complex multiplication $a \cdot b = (a_r \cdot b_r - a_i \cdot b_i) + j(a_r \cdot b_i + a_i \cdot b_r)$ requires four floating-point multiplications and two floating-point additions.

The NTT is the DFT defined over $\mathbb{Z}_q$. It follows the same concepts as the FFT, except for a different underlying field structure, $\mathbb{Z}_q$, instead of $\mathbb{C}$. It is a central building block in lattice-based cryptography as it allows faster polynomial multiplications in $\mathcal{O}(n \log n)$ time complexity than the schoolbook approach with a complexity of $\mathcal{O}(n^2)$. This allows a significant speedup for large n as used in HE settings.

The Residue Number System (RNS) representation of an element $x \in \mathbb{Z}_Q$, is a decomposition of x into smaller residuals $x_0, \ldots, x_{L-1}$, where Q is defined as in Sec. II-A. Each residual $x_i \in \mathbb{Z}_{q_i}$ is computed as $x_i = x \mod q_i$. The RNS representation allows operations on the L residues separately by using smaller integer arithmetic. The operations performed on smaller integers in the RNS domain (i.e., x_i) are reflected in the combined integer (i.e., x) which can be obtained by applying the Chinese Remainder Theorem on the residues.

C. CKKS Scheme

In 2017, Cheon *et al.* proposed a homomorphic encryption scheme, CKKS [2], which allows homomorphic computations on real numbers. Later, an efficient RNS variant of the CKKS scheme is proposed, RNS-CKKS [13]. For the rest of the paper, we use the term CKKS to refer to the RNS-CKKS.

The workflow of client-side operations in CKKS is illustrated in Fig. 1. The encoding marked in yellow first expands the input from $\mathbb{C}^{n/2}$ to $\mathbb{C}^n$ by adding each element's conjugate complex and by reordering the vector according to a dedicated pattern. Afterward, an n-point FFT is applied to the expanded vector, which generates a vector from $\mathbb{R}^n$ as the imaginary parts vanish due to the properties of expansion and FFT. A final rounding of the real parts to integers yields a vector in $\mathbb{Z}_Q[x]/(x^n + 1)$. Finally, the resulting vector is decomposed into multiple vectors/polynomials with smaller coefficients (i.e., a vector in $\mathbb{Z}_{q_i}[x]/(x^n + 1)$) using RNS.

The encoded and RNS-mapped message is then encrypted as shown in Eq. 1 for asymmetric encryption, where pk_i are the public key polynomials, m is the encoded message, and e_i and v are randomly sampled error polynomials from a centered binomial and ternary distribution, respectively. As shown in Fig. 1, the NTT representation is used to speed up polynomial multiplications.

$$(C_0, C_1) = (pk_0 \times v + e_0 + m, pk_1 \times v + e_1) \quad (1)$$

$$m' = C_1 \times sk + C_0 \quad (2)$$

Decryption, shown in Eq. 2, calculates an approximation of the message m' to be decoded, where sk refers to the secret key. Decoding is the opposite operation of encoding mapping m' to a vector of $n/2$ complex numbers by using the inverse FFT. We consider the plaintext-to-ciphertext (pt-to-ct) conversion to consist of encoding and encryption, while ciphertext-to-plaintext (ct-to-pt) refers to decryption and decoding.

III. The Proposed Design

This section presents our optimization techniques and hardware design. It focuses on the novelties of the proposed work including the FFT-based encoding, the synergy of FFT and NTT in terms of resource sharing, and the efficient RNS computation. Less attention is given to modular ring operations like NTT or error polynomial sampling, as these are already extensively covered by prior works in the literature [8], [9]. For a demonstration of our concepts, we choose the parameter $n = 8192$ which requires a corresponding Q of at most 202 bits for 128-bit post-quantum security [11]. Each modulus q_i can have from 46 up to 54 bits. Despite this parameter selection, the proposed design can easily be extended to support different parameter sets as well. For implementation and verification of CKKS client-side operations, our work closely follows the CKKS implementation in Microsoft SEAL library [14].

A. Floating-Point Unit and FFT

The CKKS scheme uses a different encoding procedure based on FFT compared to other HE schemes that operate over integer inputs, like BFV [1]. This means that CKKS requires floating-point arithmetic to perform computations during FFT.

The iterative FFT algorithm takes a vector $\mathbf{a} \in \mathbb{C}^n$ and iterates over it $\log_2(n)$ times. We refer to these iterations as stages, denoted as s_i, where $i = 0 \ldots \log_2(n) - 1$. Each stage performs a total of $n/2$ so-called butterfly operations over $\mathbf{a}$ taking two elements of $\mathbf{a}$ and one twiddle factor ω_i for computation. The computation inside the butterfly unit depends on the desired decimation method. CKKS uses DIF FFT during encoding and DIT inverse FFT during decoding. Fig. 2 gives an overview of both butterfly configurations in (a) and (b) as well as a unified version in (c). As shown on the left side both configurations contain a complex addition, subtraction, and multiplication unit. In order to save resources DIF and DIT configurations can be merged into a single architecture by adding additional multiplexers, as shown in Fig. 2 (c). Our design instantiates one such unified butterfly unit for FFT.

We accelerate the FFT by utilizing our custom floating-point unit (FPU) following the IEEE-754 standard for double-precision. In double-precision floating-point format, each number is encoded within a 64-bit word, containing one sign bit s, 11 biased exponent bits e_b, and 52 significant bits m. Based on that, the actual value v is computed as $v = (-1)^s \cdot \{1, m\} \cdot 2^{\hat{e}}$, where $\hat{e} = e_b - bias - 52$ is the unbiased exponent and $\{\cdot, \cdot\}$ indicates bit-wise appending. The FPU implementation supports addition, subtraction, and multiplication and allows several optimizations for lowering its area consumption. Multiplications of floating-point values with powers of two are just additions to the exponent. This is, for example, exploited in the decoding phase where the factor n^{-1} must be multiplied to each element. Instead of instantiating a dedicated complex multiplier, a simple adder is sufficient. Second, a complex subtractor is the same hardware circuit as an adder. Just a flip of the subtrahend's sign bit is needed. Furthermore, within the butterfly operations, the adder (yellow) and subtractor (orange) always get the same inputs, as Fig. 2 highlights. This allows sharing the floating-point preprocessing, including the denormalization of one operand, between the complex adder and subtractor. Finally, we reduce the number of registers in the butterfly module by loading its inputs just in time from the BRAM. The results are also directly stored back as soon as their computation is done without any registering. Due to the fact that each complex coefficient is 128-bit wide, a considerable amount of registers can be saved by this measure.

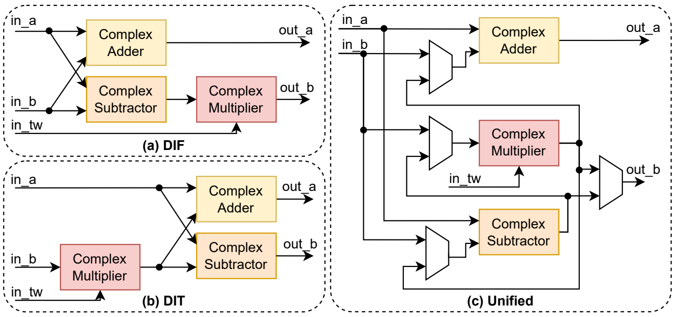

Fig. 2. (a) DIF butterfly, (b) DIT butterfly, (c) unified butterfly. Multiplexers ensure the correct connection of the subcomponents in (c) to yield configurations as in (a) and (b).

During one butterfly operation, a power of twiddle factor ω_i in $\mathbb{C}$ is multiplied with another operand. These twiddle factors depend on the polynomial size n and thus are fixed in our design. We provide the users with two design-time options for the FFT twiddle factor delivery, they are either *generated* on the fly or *stored* in memory. The choice of these options does not change Aloha-HE's latency. It just impacts resource consumption. The first option of *generated* twiddle factors requires an additional complex multiplier responsible for the generation. Thus, it introduces a higher FPGA resource utilization. However, the *stored* alternative consumes a comparably large block ROM with 128-bit wide elements for storing the needed complex-valued twiddle factors.

Furthermore, our design exploits the symmetry of the roots of unity in $\mathbb{C}$ to reduce the number of required factors from $2n$ to $n/2$. This is possible since the twiddle factors needed in the inverse transformation are the complex conjugate of the ones in the forward transformation. In addition to that, each transformation uses pairs of twiddle factors in the form of $\omega^i = a + jb$ and $\omega^{n-i} = -a + jb$, where the real part a of the complex twiddle factor is either positive or negative. Thus, the sign bit of the floating-point value needs to get flipped depending on the required twiddle factor.

B. NTT and Resource Sharing

The NTT is another important building block in CKKS. It is very similar to the FFT, the only difference is that the underlying field $\mathbb{C}$ is replaced by $\mathbb{Z}_q$. This switch from $\mathbb{C}$ to $\mathbb{Z}_q$ requires a dedicated datapath within the architecture. Yet, the overall execution flow as well as the twiddle factor order are exactly the same. This similarity allows reusing the entire control logic of FFT during NTT.

The NTT butterfly contains modular addition, subtraction, and multiplication instead of their complex counterparts in FFT. The design of modular addition and subtraction is relatively

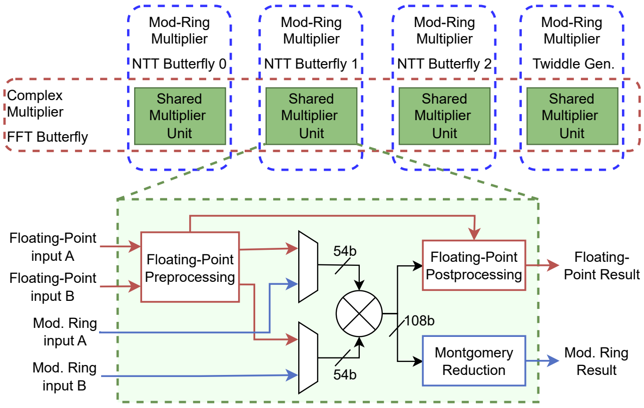

Fig. 3. Shared multiplier module. Black components are shared between the modular ring (blue) and the floating-point (red) multiplier. Four floating-point multipliers form one complex multiplier.

simple while a modular multiplier is more challenging. It requires an integer multiplication combined with a modular reduction unit. The double-precision floating-point multipliers used during FFT are internally capable of 54-bit integer multiplications. This property allows us to reuse them in our modular ring multipliers, as they require the same input bit-width.

As mentioned in Sec. II, each complex multiplication requires four floating-point multipliers. These four floating-point multipliers can share their contained integer multipliers with four modular multiplication units. We use three of them to perform three NTT transformations in parallel and one to generate the required twiddle factor. Fig. 3 gives an overview of the four shared multiplier units and their usage within either the complex or the modular multipliers. Further, it shows the architecture of the datapath within the shared multiplier. Depending on the currently executed suboperation, access to the integer multiplier (black) is granted via the multiplexers either to the FPU (red) or modular ring unit (blue). Through this optimization, we are able to save a considerable amount of 44 DSPs.

C. Adapted Word-Level Montgomery Reduction

Modular multiplication requires a modular reduction after the integer multiplication. Microsoft SEAL [14] and other related works [8], [9] use Barrett reduction [15], which requires large internal integer multipliers with inputs of up to twice the modulus size. The bit-width of the operands has a quadratic influence on DSP utilization making Barrett reduction unsuitable for low-area implementations.

The design of our reduction unit instead relies on the word-level Montgomery algorithm presented in [16] to reduce the number of DSPs. The authors exploit the prime format of NTT-friendly moduli that follow the format $q = q_H \cdot 2^w + 1$, where $w \geq \log_2(2n)$ is the so-called word size. We adapt their prime format to $q_H = 2^{k-w} - q_m$ for a k-bit prime q. This influences step 8 of Alg. 3 from [16] as shown in Eq. 3 and results in Eq. 4. This means that the $k-w$ bit-wide multiplication operand q_H of Eq. 3 is replaced by a smaller m bit-wide q_m in Eq. 4.

$$T1 = T1_H + q_H \cdot T2[w-1:0] + cin \tag{3}$$

$$T1 = T1_H + T2 \ll (k-w) - q_m \cdot T2[w-1:0] + cin \tag{4}$$

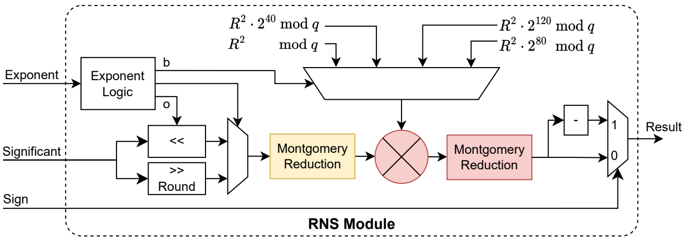

Fig. 4. Architecture of the RNS module. The yellow reduction unit is shared with the NTT Twiddle factor generation, and the red components form the modular multiplier from an NTT butterfly.

The advantage of this approach is that m can be freely chosen depending on the DSP input width. We select the parameters $w = 24$ and $m = 17$, which allows the use of a single DSP for the multiplication leading to a DSP-saving design. The disadvantage of Montgomery reduction is the introduction of a multiplicative factor R^{-1} in the reduction result. To reverse this effect, one input of each modular multiplication contains the factor R that cancels out R^{-1} after multiplication. This applies to the twiddle factors and key coefficients which are modified accordingly at design time before loading them to memory.

D. Efficient and Hardware-Friendly RNS Algorithm

The conversion of floating-point numbers to an RNS representation is another crucial step during encoding. This conversion includes rounding of potentially huge double-precision floating point numbers to integer values. The resulting integer values are then reduced by each prime q_i contained in the ciphertext modulus Q. CKKS limits Q to 202 bits in case of a polynomial degree $n = 2^{13}$ to guarantee 128-bit post-quantum security [11]. This restricts the absolute values of floating-point inputs to be smaller than 2^{201} to avoid mapping different input values to the same element of $\mathbb{Z}_Q$. Our approach benefits from the limited inputs but is not dependent on it. No additional Montgomery reduction or multiplier units are needed when larger input values should be supported.

The double-precision IEEE-754 floating-point format, as presented in Sec. III-A, is the central part of our RNS design. We use the property in Eq. 5 of the modulo operation and apply it to the IEEE-754 floating-point format as in Eq. 6.

$$a \cdot b \ \% \ q = (a \ \% \ q) \cdot (b \ \% \ q) \ \% \ q \tag{5}$$

$$\{1, m\} \cdot 2^{\hat{e}} \ \% \ q = (\{1, m\} \ \% \ q) \cdot (2^{\hat{e}} \ \% \ q) \ \% \ q \tag{6}$$

$$= (\{1, m\} \ll o \ \% \ q) \cdot (2^{b} \ \% \ q) \ \% \ q \tag{7}$$

It splits up the reduction of the large floating-point word into two smaller ones. Their results are then multiplied and again reduced. However, the 53-bit wide significant $\{1, m\}$ does not fully fill the Montgomery reduction unit's input of $2k$ bits, where $k = 46$ is the bit-width of the smallest supported modulus. Therefore, the overall operation can further be improved as in Eq. (7). The parameters o and b follow the relation $o + b = \hat{e}$, while $o + 53 \leq 2k$ must hold to not exceed the Montgomery units bit-width. We further select $b = i(2k - 53 + 1)$ with $i \in \mathbb{N}_0$ in a way such that a minimal number of distinct values for b are needed to fully cover the possible input range. This number depends on k, the maximum bit-size of Q, and the bit-width of the significant. For our selected parameter set, four values are required.

These observations are used for our hardware-friendly architecture of Aloha-HE, illustrated in Fig. 4. The exponent logic first tests whether $\hat{e}$ is negative. In this case, b is 0, and a right shift followed by rounding of the significant is performed. Otherwise, b and o are given by $\hat{e}$ as previously described and the significant gets shifted left by o. Afterward, the corresponding intermediate result enters a Montgomery reduction unit (yellow). The value of b selects one of four precomputed constants in form of $2^b \cdot R^2 \ \% \ q$ stored in a small ROM. It contains the multiplicative factor R^2 to compensate for the two Montgomery reductions and gets modulo-multiplied to the first reduction's output (red). Finally, a conditional additive inversion depending on the sign bit yields the overall result.

The proposed architecture allows rounding and reduction of any valid floating-point input in constant time. Its datapath is fully pipelined and can be easily extended to support single precision floating point formats or different modulus sizes.

E. Overall Design

Aloha-HE is designed as a co-processor for client-side operations in homomorphic encryption. The overall architecture together with the main components is illustrated in Fig.5. AXI4 bus connections shown on the top left establish the interface to the host CPU and main memory. Instructions, operands, and status information are exchanged directly between the CPU and accelerator, while the transfer of the large polynomials is handled by direct memory access (DMA). The DMA controller is configured via a second AXI4 port and streams the desired polynomials directly from RAM into the co-processor's BRAM set without consuming CPU time.

The core element in Aloha-HE is the shared arithmetic unit marked in grey. It contains the datapath of the four shared integer multipliers and Montgomery reduction units. Besides sharing of multipliers between floating-point and modular ring multipliers as discussed in Sec. III-B, additional resource sharing is possible. The RNS unit makes use of two Montgomery reduction units and one integer multiplier. Furthermore, the two point-wise multiplications shown in Eq. 1 of Sec. II are performed concurrently using two of the existing NTT butterflies. This high re-utilization of existing hardware resources leads to low idle times of components and thus to an efficient design.

IV. Results

This section gives an overview of relevant prior works, presents our results, and a comparison with prior works.

A. Prior Works

There are a few works in the literature targeting the implementation and acceleration of client-side CKKS operations on low-end devices. SEAL-Embedded [6] is a software library that adapts the encoding and encryption of Microsoft SEAL [14] for memory-constrained embedded systems. These systems are expected to only encrypt and send data to the server. Thus, no decryption functionality is provided by SEAL-Embedded, which is a limitation of possible application scenarios of this approach. Azad *et al.* builds up on SEAL-Embedded and present RACE [7], an ASIC co-processor for modular ring operations including NTT and point-wise multiplication. They later extended this work to a newer version, named RISE [8]. The RISE extends the RACE with an error sampling unit and additional NTT butterfly units, and further speeds up encryption. Both works still perform the encoding procedure in software. This limits the achievable speedup and introduces the need for floating-point support on the host CPU. The work in [9] presents an FPGA-based accelerator to perform symmetric encryption in the SEAL-Embedded library. Similar to the RISE, hardware support for modular ring operations and error sampling is provided, however, the encoding is performed in software. This leads to the same limitations as in [7] and [8].

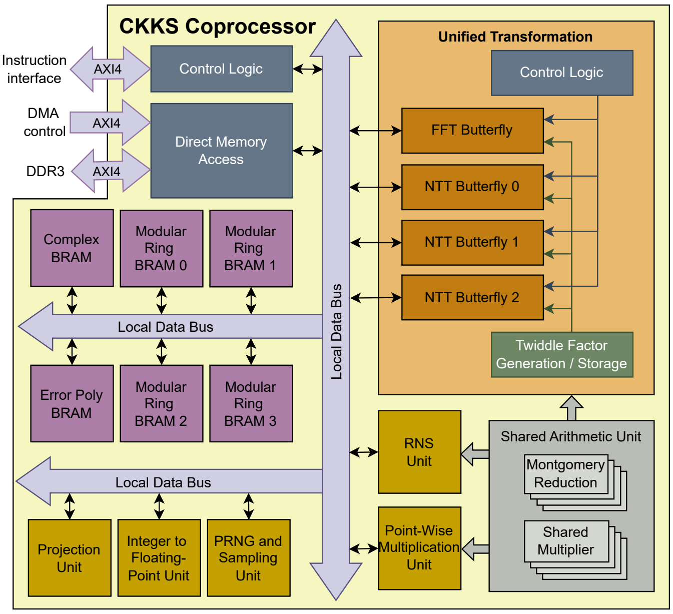

Fig. 5. Overview of Aloha-HE's hardware architecture.

B. Implementation Results and Comparison

We use Xilinx Vivado 2019.1 to synthesize and implement our design, Aloha-HE. The resulting hardware is verified on actual hardware with two different low-cost FPGAs, namely Kintex-7 FPGA (Genesys2, xc7k325tffg900-2) and ZYNQ-7000 SoC (pynq-z2, xc7z020clg400-1). The ZYNQ architecture provides a dual-core ARM CPU running at 650MHz. The Aloha-HE is instantiated on the FPGA in ZYNQ SoC and runs at 130MHz. On the Kintex-7 FPGA, the soft CPU and Aloha-HE reside on the FPGA and run at 200MHz. As the soft CPU, we use MicroBlaze softcore IP from Xilinx. In Table I, we present area and performance results of our work (for two platforms) and the other works in literature targeting client-side operations of CKKS. Note that our performance results, measured using actual FPGAs, present end-to-end execution time for CKKS pt-to-ct and ct-to-pt operations. These overall execution times cover all required software, hardware, and data transfer latency in the complete system.

Compared to the hardware accelerators RISE [8] and RACE [7], Aloha-HE on the Kintex-7 FPGA improves the

TABLE I
IMPLEMENTATION RESULTS AND COMPARISON WITH RELATED WORK

Work	Platform	n	$\log Q$	Area				Latency (in ms)				
				LUT	REG	DSP	BRAM	FFT	RNS	NTT	Encr.	Decr.
SEAL [14]	Intel i5-8250U @2.4GHz	8192	3×54 bit	-				-	-	0.135	4.91	0.73
SEAL-Embed. [6]	ARM Cortex-A7 @494MHz	4096	3×30 bit	-				-	-	3.221	91.27	-
RACE [7]	ASIC 12nm @1GHz	8192	130 bit	63,788 μm^2				-	-	-	110.28	19.08
RISE [8][b]	ASIC 12nm @1GHz	8192	180 bit	113,705 μm^2				-	-	-	20	19
		4096	30 bit	88,120 μm^2				-	-	0.002	-	-
[9]	Zynq Ultrascale+ @150MHz	8192	32 bit	-	-	-	-	-	-	-	7.79	-
		4096	32 bit	3320[a]	1480[a]	42[a]	29.5[a]	-	-	0.137	-	-
Our[c]	Kintex-7 @200MHz	8192	3×54 bit	17680	14431	56	97	0.267	0.041	0.267	1.87	0.87
Our[d]	Kintex-7 @200MHz	8192	3×54 bit	20728	17647	100	82.5	0.267	0.041	0.267	1.87	0.87
Our[c]	ZYNQ-7000 @130MHz	8192	3×54 bit	19274	15163	56	97	0.410	0.064	0.410	3.04	1.30
Our[d]	ZYNQ-7000 @130MHz	8192	3×54 bit	20253	18152	100	82.5	0.410	0.064	0.410	3.04	1.30

[a] Utilization for NTT unit only. [b] Implementation results just presented in diagrams. No exact values are available.
[c] Our implementation using stored FFT twiddle factors. [d] Our implementation using generated FFT twiddle factors.

encryption latency by 10.7× and 59×, respectively. Although these ASIC designs operate at 1GHz, Aloha-HE running at 200MHz outperforms them significantly by additionally overtaking the encoding from the CPU. The FPGA design in [9] reports performance results for encryption and hardware utilization figures only for the NTT unit, making a direct comparison hard. Aloha-HE shows an improvement in latency of 4.2× compared to [9]. Although the Aloha-HE variant storing FFT twiddle factors uses 33% more DSPs, Aloha-HE supports larger moduli up to 54 bits while the design in [9] targets 32-bit moduli. Finally, our work is compared with the software libraries SEAL [14] and SEAL-Embedded [6]. Compared to the SEAL library running on a high-end Intel Core i5 CPU, Aloha-HE allows up to 2.6× faster pt-to-ct conversion for the same parameter set. For ct-to-pt conversion, our work shows slightly lower performance as the number of operations that can be accelerated during the decryption operation is low. Moreover, data transfer overhead prevails over the latency of the operation in FPGA for ct-to-pt conversion. However, the SEAL library running on high-end CPUs is not suitable for area-constrained or embedded systems. Compared to the SEAL-Embedded library, our work shows up to 49× speedup for pt-to-ct conversion, although our design supports larger parameters.

We finally consider the latency of single NTT operations shown in Tab. I. Related works such as SEAL-Embedded, [9], and RISE just report NTT benchmarks for smaller $n = 4096$, which hardens comparison. Aloha-HE still shows a 35% and 97% better NTT performance compared to [9] and SEAL-Embedded respectively as it performs three concurrent NTT transformations in 267μs. Yet, RISE uses an NTT with 32 butterflies and shows 44.5× better NTT performance. Compared to RISE, Aloha-HE uses only three butterflies and still reaches 10.7× better performance for entire pt-to-ct operations.

V. CONCLUSION

In this paper, we present Aloha-HE, the first hardware accelerator supporting the whole client-side operations for the CKKS scheme, following Microsoft SEAL implementation. Aloha-HE benefits from a high degree of resource sharing between similar FFT and NTT execution and data flows. Moreover, it uses our novel floating-point-based RNS algorithm. Despite targeting a resource-saving approach tailored for low-end hardware platforms, Aloha-HE reduces the latency of the client-side operations by a factor of up to 59× compared to the existing CKKS hardware accelerators. Our presented methodology can easily be extended to various parameter sets or even different HE schemes, which can allow the adoption of Aloha-HE in a broad field of applications in the encrypted computation domain.

REFERENCES

[1] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012.

[2] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[3] B. Jacobs, "Data protection in virtual environments (dprive)," Defense Advanced Research Projects Agency (DARPA), 2023, https://www.darpa.mil/program/data-protection-in-virtual-environments.

[4] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *2019 22nd Euromicro Conf. on Digital System Design (DSD)*, 2019, pp. 253–260.

[5] B. Reagen *et al.*, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 26–39.

[6] D. Natarajan and W. Dai, "Seal-embedded: A homomorphic encryption library for the internet of things," *IACR TCHES*, vol. 2021, no. 3, p. 756–779, Jul. 2021.

[7] Z. Azad *et al.*, "Race: Risc-v soc for en/decryption acceleration on the edge for homomorphic computation," in *ACM/IEEE ISLPED'22*, 2022.

[8] ——, "Rise: Risc-v soc for en/decryption acceleration on the edge for homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2023.

[9] S. D. Matteo, M. L. Gerfo, and S. Saponara, "Vlsi design and fpga implementation of an ntt hardware accelerator for homomorphic seal-embedded library," *IEEE Access*, vol. 11, 2023.

[10] "Aloha-he source code," https://github.com/flokrieger/Aloha-HE.

[11] M. Albrecht *et al.*, "Homomorphic encryption standard," Cryptology ePrint Archive, Paper 2019/939, 2019.

[12] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," Cryptology ePrint Archive, Paper 2016/504, 2016.

[13] J. H. Cheon *et al.*, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 2019, pp. 347–368.

[14] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan 2023, microsoft Research, Redmond, WA.

[15] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[16] A. C. Mert *et al.*, "A flexible and scalable ntt hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *DATE'20*, 2020, pp. 346–351.