A Survey on Several Hardware Accelerators for Fully Homomorphic Encryption

XXX
XXX
XXX
XXX
XXX

***Abstract*—Fully Homomorphic Encryption (FHE) has experienced significant theoretical advancements and continuous breakthroughs recently, driving its widespread application in various privacy-preserving scenarios. However, due to its high computational overhead and storage costs, the practical application of FHE remains limited. To address these challenges, researchers have proposed a variety of efficient acceleration solutions.**

In this paper, we provide a comprehensive survey on several hardware accelerators designed for FHE, mainly focusing on the CKKS and TFHE schemes. First, we present a detailed breakdown and explanation of the algorithms utilized in the CKKS and TFHE scheme. Next, we make an in-depth analysis of the evolution of design details and hardware-based optimizations from primitive-level operations to function-level operations. Finally, We summarize all these optimizations made so far and explore the future research directions of hardware accelerators for FHE. We identify various potential area for development, such as exploring advanced algorithms, designing novel hardware architectures, selecting the most suitable parameters for practical applications and so on. By highlighting these potential research directions, we aim to encourage further advancements in hardware accelerators for FHE and its application to various fields for privacy preservation.

***Index Terms*—Fully homomorphic encryption, CKKS, TFHE, Hardware accelerator**

I. Introduction

As an encryption scheme, homomorphic encryption (HE) is an encryption scheme that allows computations to be performed on encrypted data without decrypting it first. This means that sensitive data can remain encrypted during processing, ensuring that privacy is maintained even when the data is processed by an untrusted entity. The primary HE can perform operations such as addition or multiplication on encrypted data, which makes it valuable in fields like cloud computing, secure data analysis and machine learning [1]–[3].

Homomorphic encryption computation can be classified into arithmetic homomorphism and logical homomorphism based on its load characteristics. The examples of algorithms and development history of HE are shown in Fig. 1. Arithmetic homomorphism supports large-scale computation on integers or real numbers, mainly focusing on linear functions. Among them, BGV [4] and BFV [5] can perform modular operations over finite fields, where modulus is typically a prime number or a prime power. The decryption results are relatively accurate compared to the plaintext results. Besides, CKKS [6] supports approximate computation on real and complex numbers and allows for slight precision loss in the results. On the other hand, logical homomorphism can perform bitwise operations and nonlinear functions, primarily based on Boolean circuits and decision graphs. The DM and CGGI are two main schemes, which were initially implemented in the FHEW [7] and TFHE [8] libraries.

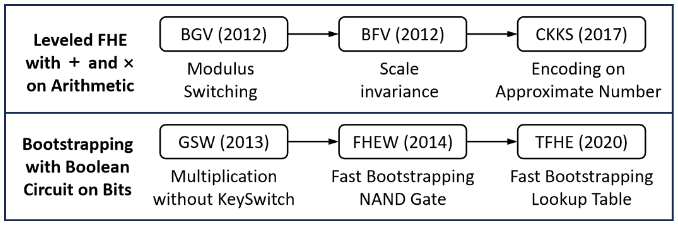
Leveled FHE with + and × on Arithmetic
BGV (2012)
Modulus Switching
BFV (2012)
Scale invariance
CKKS (2017)
Encoding on Approximate Number
Bootstrapping with Boolean Circuit on Bits
GSW (2013)
Multiplication without KeySwitch
FHEW (2014)
Fast Bootstrapping NAND Gate
TFHE (2020)
Fast Bootstrapping Lookup Table

Fig. 1. Examples of arithmetic and logical homomorphism

While HE offers strong privacy guarantees, it comes with a set of challenges. Performing computations on encrypted data is more resource-intensive than working with unencrypted data. For example, An application that is only 30.5 Kb in plaintext state dramatically expands to 1.33 Gb in ciphertext state. Besides, the time required for computation increase from around 0.1 to 390 seconds. Moreover, the complexity of HE operations often causes higher energy consumption, making it unsuitable for resource-constrained environments.

Given the computational and efficiency challenges of FHE, there is an increasing demand for hardware accelerators. Hardware acceleration such as GPUs, FPGAs and ASICs, can provide a promising approach to improving the overall performance. Hardware accelerators equipped with specialized processing power can effectively address the computational and storage challenges of FHE, making it more applicable for real-world scenarios.

In this paper, we make a comprehensive survey on several hardware accelerators for fully homomorphic encryption by the end of March 2025, especially for the CKKS and TFHE schemes. The main contributions of our work are as follows.

- To make homomorphic encryption easier to understand, we present a detailed breakdown and explanation of the algorithms utilized in the CKKS and TFHE scheme.
- From primitive-level operations to function-level operations, we make an in-depth analysis of the evolution of design details and hardware-based optimizations.

- Finally, We summarize all these optimizations made so far and explore the future research directions of hardware accelerators for homomorphic encryption. We identify various potential area for development, such as exploring advanced algorithms, designing novel hardware architectures, selecting the most suitable parameters for practical applications and so on. By highlighting these potential research directions, we aim to encourage further advancements in hardware accelerators for FHE and its application to various fields for privacy preservation.

II. Background

In this section, we describe the data types, primitive-level operations and function-level operations utilized in the CKKS and TFHE scheme. To enhance clarity, parameters and notations are summarized in Table I.

TABLE I
NOTATIONS AND PARAMETERS

Notation	Description
$\mathbf{m}$	Plaintext vector.
$[[\mathbf{m}]]$	RLWE ciphertext of $\mathbf{m}$. $[[\mathbf{m}]] = (a(X), b(X))$.
m	Scalar plaintext.
$[[m]]$	LWE ciphertext of scalar message m. $[[m]] = (a, b)$.
$P(X)$	Polynomial.
$P_{\mathbf{m}}$	Plaintext polynomial corresponding to message $\mathbf{m}$.
n	The number of slots in a ciphertext.
N	The polynomial size.
Δ	The scale factor.
R_Q	The polynomial ring, $R_Q = \mathbb{Z}_Q/(X^N + 1)$.
Q	The polynomial modulus $Q = \prod_{i=0}^{L} q_i$.
P	The special prime $P = \prod_{i=0}^{\alpha} q_i$.
q_i	The small RNS moduli composing Q.
p_i	The small RNS moduli composing P.
L	The maximum level of polynomial.
l	The current level of polynomial.
R_{q_l}	Polynomial at level l.
d_{num}	The decomposition number.
α	The number of RNS moduli, $\alpha = \lceil (L+1)/d_{num} \rceil$.
β	$\beta = \lceil (l+1)/d_{num} \rceil$.
evk	The evaluation key.
n_{lwe}	The dimension of LWE ciphertext.
k	The dimension of GLWE ciphertext.
q	The modulus of coefficients.
l_b	The decomposition level of Bootstrapping Key.
l_k	The decomposition level of TFHE KeySwitching Key.
ACC_i	The accumulation ciphertext in i-th iteration.
ksk	The KeySwitching Key in TFHE.
bsk	The Bootstrapping Key in TFHE.
n_{slot}	The number of valid slots in the RLWE ciphertext.

A. CKKS scheme

CKKS is a kind of arithmetic homomorphic encryption, where the data (usually n number of real or complex numbers) is first packed into a vector message $\mathbf{m}$ (see in Eq. 1).

$$\mathbf{m} = (m_0, m_1, \cdots, m_{n-1}) \in \mathbb{C}^n \quad (1)$$

Encode. The message $\mathbf{m}$ is encoded as a polynomial $P_{\mathbf{m}}$ after a series of operations consisting of conjugate extension $\rightarrow$ projection $\rightarrow$ multiplication $\rightarrow$ rounding (see in Eq. 2). Due to the rounding operation, each coefficient of $P_{\mathbf{m}}$ contains small errors.

$$\lceil \Delta \cdot \sigma^{-1}(\pi^{-1}(\mathbf{m})) \rfloor \mapsto P_{\mathbf{m}} = a_0 + \cdots + a_{N-1}X^{N-1} \quad (2)$$

Encryption. The ciphertext $[[\mathbf{m}]]$ is obtained after encrypting the plaintext $P_{\mathbf{m}}$, where $A_{\mathbf{m}}$ is a random large-coefficient polynomial, S is a secret polynomial, and E is a small-coefficient error polynomial (see in Eq. 3).

$$[[\mathbf{m}]] = (B_{\mathbf{m}}, A_{\mathbf{m}}) = (A_{\mathbf{m}} \times S + P_{\mathbf{m}} + E, A_{\mathbf{m}}) \in \mathcal{R}_Q^2 \quad (3)$$

Function-level operations. $[[\mathbf{m}]]$ can be sent to a server to offload computation on $\mathbf{m}$. Then the server performs a homomorphic evaluation $f()$ on $[[\mathbf{m}]]$, which consists of a series of function-level operations, such as addition and multiplication between $[[\mathbf{m}]]$ and another ciphertext (HAdd, HMult) or a plaintext (PAdd, PMult), and cyclic rotation of $[[\mathbf{m}]]$ (HRotate). In order to support HMult and HRotate, we introduce KeySwitching to make ciphertext decryptable with the original key. This operation is composed of Base Conversion (BConv) and Inner Product (IP). BConv is used to change the prime limb set of a polynomial in the RNS variant of the CKKS scheme [9]. IP is used to multiply a polynomial with the evaluation key (evk) during KeySwitching. In terms of noise management, HRescale divides the ciphertext by $\triangle$ to recover its scale back to $\triangle$ [10]. Bootstrapping is used to reduce noise in encrypted computation, restoring the ciphertext's computational capacity to support more function-level operations. These function-level operations can be implemented by combining several primitive-level operations, as shown in Table II and Fig. 2.

TABLE II
DECOMPOSITION OF FUNCTION-LEVEL OPERATIONS INTO PRIMITIVE-LEVEL OPERATIONS

Scheme	Operation	Primitives
CKKS	PAdd	ModAdd.
	HAdd	ModAdd.
	PMult	ModAdd, ModMult.
	HMult	ModAdd, ModMult, NTT
	HRotate	ModAdd, ModMult, NTT, Automorphism.
	HRescale	ModAdd, ModMult, NTT.
	KeySwitching	ModAdd, ModMult, NTT.
	Bootstrapping	ModAdd, ModMult, NTT, Automorphism.
TFHE	KeySwitching	ModAdd, ModMult, Reduction.
	Bootstrapping	ModAdd, ModMult, NTT, Decomposition, Rotation, Extraction.

Primitive-level operations. (1) **ModAdd** is used to perform element-wise modular addition of two polynomials. (2) **ModMult** is used to perform element-wise modular multiplication of two polynomials that are in evaluation representation. (3) Number Theoretic Transform (**NTT**) [11] is used to accelerate polynomial multiplication by transforming a polynomial from coefficient representation to evaluation, reducing the time complexity to $O(NlogN)$. (4) **Automorphism** is used to move the i-th coefficient of a polynomial to the position of the $\psi(i)$-th coefficient by applying the mapping to each limb.

B. TFHE scheme

Fast fully homomorphic encryption scheme over the torus (TFHE) [8] is a kind of logic homomorphic encryption, which enables Bootstrapping to be performed in under 0.1 seconds using a single-core CPU during the evaluation of arbitrary univariate functions [7], [12].

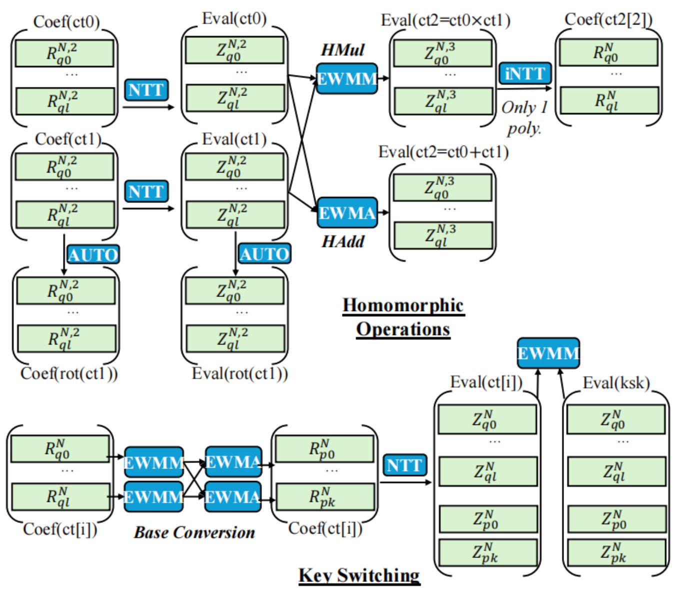

Fig. 2. Decomposition of the CKKS scheme

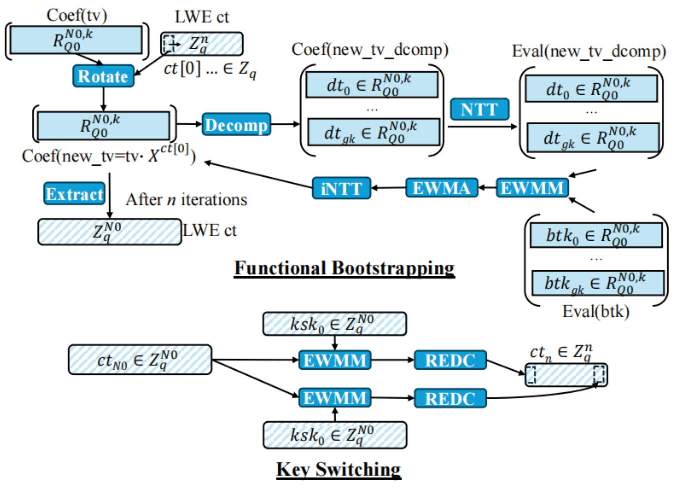

Fig. 3. Decomposition of the TFHE scheme

There are three types of ciphertexts in TFHE, namely LWE ciphertext, Genenral-LWE (GLWE) ciphertext and Genenral-GSW (GGSW) ciphertext.

LWE ciphertext. LWE ciphertext is used to encrypt the scalar type of message m. If we define n as the dimension of the LWE ciphertext, p as the modulus of plaintext, the secret key $\mathbf{s} = (s_1, \cdots, s_n) \in \mathbb{B}^n = \{0,1\}^n$, then the LWE ciphertext $\mathbf{c} = \text{LWE}_s(m) = (a_0, \cdots, a_n, b) \in \mathbb{T}_q^{n+1}$. In this case, $(a_0, \cdots, a_n)$ is the LWE mask and $b = \sum_{i=1}^{n} a_i \cdot s_i + m + e$, where e is a small noise term. In this implementation, LWE ciphertext can be described as a scalar of size $(n + 1)$.

GLWE ciphertext. GLWE ciphertext is used to encrypt the polynomial type of message m. If we define k as the dimension of the GLWE ciphertext, and the secret key of the polynomials $\mathbf{S} = (S_1(x), \cdots, S_k(x)) \in \mathbb{B}_N[X]^k$, then the GLWE ciphertext $\mathbf{C} = \text{GLWE}_\mathbf{S}(m) = (A_1(x), \cdots, A_k(x), B(x)) \in \mathbb{T}_{(q,N)}[X]^{k+1}$. In this case, $(A_1(x), \cdots, A_k(x))$ is the GLWE mask similarly and $B(x) = \sum_{i=1}^{k} A_i(x) \cdot S_i(x) + M(x) + E(x)$, where $E(x)$ is a small noise term. In this implementation, GLWE ciphertext can be described as a vector of polynomials of size $(k + 1)$.

Algorithm 1 TFHE PBS (c, tv, bsk, ksk)

```
Input: $c$: a LWE ciphertext, $c = (\mathbf{a}, b)$; $tv$: the test vector; $bsk$: the bootstrapping key; $ksk$: the key switching key
1: $\tilde{c} = (\tilde{\mathbf{a}}, \tilde{b}) = \text{ModSwitch}(c)$ // ModSwitch
2: $\text{ACC}_0 = \text{Rotate}(tv, b)$
3: // Blind Rotation
4: for $i = 1$ to $n_{\text{lwe}}$ do
5:     $\text{tmp} = (\text{Rotate}(\text{ACC}_{i-1}, \tilde{a}_i) - \text{ACC}_{i-1})$
6:     $\text{tmp} = \text{Decompose}(\text{tmp}, l_b)$
7:     // External Product
8:     for $j = 1$ to $(k+1)l_b$ do
9:         $\text{tmp\_acc} = \text{tmp\_acc} + \text{NTT}(\text{tmp}[j]) * \text{bsk}[i][j]$
10:    end for
11:    $\text{ACC}_i = \text{ACC}_{i-1} + \text{INTT}(\text{tmp\_acc})$
12: end for
13: // SampleExtract
14: $c' = (\mathbf{a}', b) = (a'_1, \ldots, a'_{kN}, b')$
15: $= \text{SampleExtract}(\text{ACC}_n, 0)$
16: // TFHE KeySwitching
17: $\mathbf{a}'' = \text{Decompose}(\mathbf{a}', l_k)$
18: $c'' = (0, \ldots, 0, b') - \sum_{i=0}^{kN} \sum_{j=0}^{l_k} a''_i[j] * \text{ksk}[i][j]$
19: return $c''$
```

GGSW ciphertext. GGSW ciphertext is the extension of the GLWE ciphertext. If we define l as the decomposition level, and β as the decomposition base, then the GGSW ciphertext $\mathbf{C} = \text{GGSW}_\mathbf{S}(m) \in \mathbb{T}_{(q,N)}[X]^{(k+1)\times l\times(k+1)}$. In this implementation, GGSW ciphertext can be described as a matrix of polynomials of size $(k + 1) \times l \times (k + 1)$.

Function-level operations. Programmable Bootstrapping (**PBS**) is the most critical operation in TFHE, which can be used to construct any function, such as logical and relational operations. After PBS, **KeySwitching** is used to make ciphertext decryptable with the original key. Similar to the CKKS scheme, these function-level operations can be implemented by combining several primitive-level operations, as shown in Table II and Fig. 3.

Primitive-level operations. Compared to CKKS, TFHE requires additional computing cores. (5) **Decomposition** is used to decompose one polynomial into smaller polynomials. (6) **Rotation** is used to carry out negacyclic rotation and polynomial subtraction. (7) **Extraction** is used to form the new LWE ciphertext by extracting a specific coefficient from the GLWE ciphertext. (8) **Reduction** is used to transform the LWE ciphertext to the scalar.

We can combine these primitive-level operations to perform different function-level operations. Although these primitive-level operations can be fully implemented in software, their performance fails to meet practical requirements due to inherent limitations of general-purpose processors. Sequential execution of modular arithmetic, memory-intensive polynomial operations, and lack of hardware-level parallelism lead to significant latency.

For example, a single HMult may require milliseconds on CPUs, while complex neural network inferences require seconds or even minutes. To bridge this gap, dedicated hardware acceleration becomes a feasible solution. By tailoring arithmetic units, memory hierarchies, and dataflow pipelines specifically for these primitive-level operations, we can achieve orders-of-magnitude speedups, enabling real-world deployment in latency-sensitive applications such as private inference and real-time secure analytics.

III. Computing cores with primitive-level optimizations

A. ModAdd and ModMult unit

ModAdd is used to perform element-wise addition, followed by the modulo operation to the result. Since each input polynomial has already undergone modular reduction, its value will not be greater than the modulus q. Therefore, the algorithm of ModAdd is shown in Equation 4.

$$(a+b) \bmod q = \begin{cases} a+b, & a+b<q \\ a+b-q, & a+b \geq q \end{cases} \tag{4}$$

Compared with ModAdd, ModMult is more complex, as seen in Equation 5. Given two polynomials a and b, multiplying a with b obtains the partial result res. The res will be divided by the modulus q to obtain the quotient p. Then, the res subtracted by $(p \times q)$ is the $result$.

$$\begin{cases} res = a \times b \\ p = res/q \\ result = res - (p \times q) \end{cases} \tag{5}$$

Shivdikar et al. [13] propose a single-stage fused polynomial multiplication, which offers significant speedup for multiplying two polynomials at minimized cost for multiplying larger numbers of polynomials. They use Karatsuba algorithm [14] to reduce the number of modular products by $N/2$ compared to the single-stage algorithm of [15].

Since implementing division is expensive, researchers have attempted to optimize the process. Prior work [84] employs a Montgomery multiplication [85] and recognizes that it can be improved if an appropriate modulus is chosen. Samardzic et al. [16] (F1) only selects NTT-friendly moduli q such that $q \equiv -1 \bmod 2^{16}$, thus removing a multiplier stage from [84], reducing area by 19% and power by 30%.

Based on F1, since multiplier area and power scale quadratically with the bitwidth, Samardzic et al. [17] (CraterLake) adopt a narrower datapath (28-bit for CraterLake while 32-bit for F1) to improve the high overhead and ensure enough NTT-friendly moduli. Secondly, they pipeline each multiplier to its energy-optimal point. These approaches improve area per bit by 1.6× and energy per bit by 1.3× over F1.

Besides, Yang et al. [10] (Poseidon) employ Barrett modular reduction [18]–[21] for the division result to avoid the high overhead.

Kim et al. [22] (SHARP) design a versatile element-wise engine (EWE) that supports five instructions covering all element-wise operations in the CKKS scheme. Specifically, an EWE, as a centralized structure, is composed of four Barrett modular multipliers, two adders, input and output buffers and a control unit.

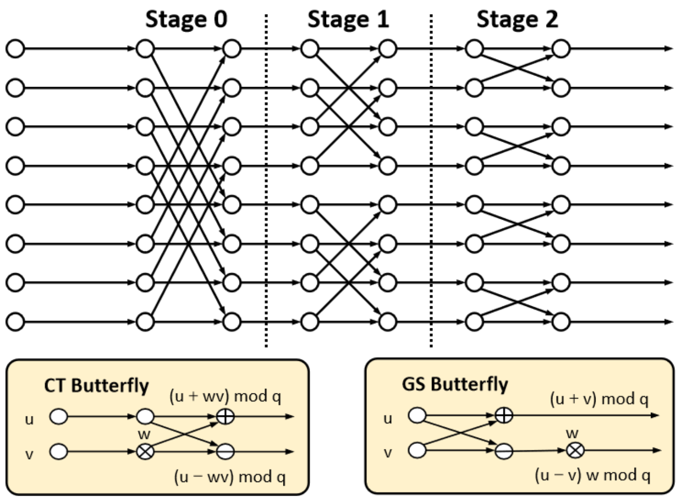

Fig. 4. NTT unit structure

B. NTT unit

NTT is used to speed up polynomial multiplication by transforming a polynomial from its coefficient representation to the evaluation representation. The structure of the NTT is shown in Fig. 4. For simplicity, we provide an example of 8-point polynomial NTT or INTT and butterfly operations. It should be noted that NTT uses CT butterfly, while INTT uses GS butterfly [23].

Samardzic et al. [16] (F1) propose the four-step variant of the FFT algorithm [24], adding an extra multiplication to produce a vector-friendly decomposition. It performs a series of DIT NTT $\rightarrow$ Multiply $\rightarrow$ Transpose $\rightarrow$ DIF NTT, where the quadrant-swap Transpose unit transposes an $E \times E$ matrix by recursively decomposing it into quadrants. Crucially, F1 can support all values of E using a single four-step NTT pipeline by conditionally bypassing layers in the second NTT butterfly.

Although the NTT units in F1 are effective for a small number of lanes, Samardzic et al. [17] (CraterLake) expand it to 256 lanes, causing severe data communication overhead when exchanging data across lanes.

NTT suffers from severe main-memory bandwidth bottleneck on large HE parameter sets. To tackle the main-memory bandwidth issue, Kim et al. [25] propose a novel NTT-specific on-the-fly root generation scheme dubbed on-the-fly twiddling, achieve 4.2× speedup on a modern GPU.

Observing that twisting factors show geometric progression, Kim et al. [11] (ARK) devise on-the-fly twisting factor generation on NTT units, nearly eliminating the memory traffic for loading twisting factors.

In order to reduce the high communication overhead in NTT, Kim et al. [22] (SHARP) devise a ten-step NTT unit along with a corresponding hierarchical structure. Regarding an input limb as an $M^2 \times M^2$ matrix for $M = \sqrt[4]{N}$. Initially, the first -point four-step NTT phase is performed. These results are stored in an NTT unit buffer and are read from the buffer in a different order after all columns arrive. An inter-lane-group Transpose is performed through a direct cluster-wide wire connection, which is the only semi-global connection required in NTT unit design. Finally, the second four-step NTT phase is performed. Overall, the hierarchical architecture of SHARP is a solution scalable to cluster designs with numerous (256) lanes utilized in previous vector FHE accelerators. This results in a six-fold reduction in the horizontal bisection bandwidth of an NTT unit and a 9.17× decrease in wiring length for horizontal connections compared to ARK.

C. Automorphism unit

Automorphism moves the i-th coefficient of a polynomial to the position of the $\psi(i)$-th coefficient by applying the mapping in Equation 6 to each limb. We reuse the notation to denote Automorphism on a polynomial P_m as $\psi_r(P_m)$ and refer to r as the rotation amount.

$$\psi_r : i \mapsto (i \cdot 5^r) \bmod N, \quad i = 0, 1, \ldots, N-1 \tag{6}$$

Automorphism poses significant difficulties for the hardware design due to the large scale of polynomial degree N and the N-based coordinate mapping.

Yang et al. [10] (Poseidon) segment the N-element vector into several C-element sub-vectors to reduce the mapping domain. In this case, the mapping of I-coordinates can be achieved by the row mapping twice, while the mapping of J-coordinate can be achieved by the column mapping.

Samardzic et al. [16] (F1) utilize a Transpose unit to perform permutation operation on the matrix, transforming row transformation to column transformation.

Zhou et al. (UFC) exploit the basic theory of NTT and reuse NTT interconnect to process Automorphism by setting the appropriate root of the unit for computation. Considering Automorphism is followed by BConv for KeySwitching, which ensures polynomial in the coefficient form. As such, the replacement requires extra NTT and INTT operation.

D. Decomposition unit

Decomposition unit is used to decompose input polynomials into several smaller value polynomials. Decomposition unit receives a polynomial as input and decomposes it into l_b polynomials. However, this decomposition differs from the commonly used RNS decomposition in the CKKS scheme. The decomposition process for each coefficient in the polynomial is carried out according to the following equation.

$$\left\| a - \sum_{i=1}^{l_b} \left\lfloor \frac{a}{Q/B^i} \right\rfloor_B \right\|_\infty \leq \frac{Q}{B^{l_b}} \tag{7}$$

In Equation 7, a is the coefficient of the polynomial, Q is the original ciphertext modulo, B is the new basis for decomposition, and l_b is the decomposition level.

The challenge lies in designing a fully pipelined Decomposition unit that can accept input and produce output every cycle, thereby maintaining maximum throughput. Additionally, the Decomposition process must be executed for each coefficient in the polynomial. As evident, the required division and rounding operations are quite complex and expensive.

To address this issue, Putra et al. [26] (Strix) propose a novel Decomposition unit that divides the process into two steps: rounding and extraction. In order to mitigate the high costs, they implement the entire circuit without using any multipliers. By employing a combination of masking, shifting, and adding, both the circuit cost and overall complexity are significantly reduced.

IV. Hardware accelerators with function-level optimizations

To balance the computational load, we can combine these primitive-level operations to perform a series of function-level operations. Through reasonable scheduling, we can effectively reduce both computational overhead and storage costs in that scheduling not only optimizes the computation order but also maximizes hardware utilization, avoiding unnecessary resource waste. Based on this, multiple optimization schemes tailored to different accelerations have been proposed. These optimization schemes adopt customized scheduling strategies and data access patterns based on the characteristics of each platform to further enhance performance.

A. GPU-based accelerators

GPUs excel in parallelizing operations, making them well-suited for CKKS schemes. With thousands of cores, GPUs are able to process large numbers of operations such as matrix or vector multiplications at the same time. Moreover, GPUs also allow for efficient tensor operations, which are often necessary in encrypted data analytics and machine learning models. Several researches evaluate various FHE schemes on different GPU platforms to identify the most suitable architecture, leveraging the ample bandwidth provided by high-end GPUs.

For the CKKS scheme, Al Badawi et al. [27] first propose PrivFT, a GPU implementation of leveled CKKS scheme that shows 1 to 2 orders of magnitude speedup compared to CPU. An important optimization is related to matrix packing, which can reduce the number of HE operations and ciphertexts.

Jung et al. [28] conduct a rigorous analysis of the computational complexity and data access patterns of HMult. On a GPU, they utilize massive thread-level parallelism expressible through CUDA. The two main optimizations are (1) matrix transposition to exploit access locality and (2) loop reordering to expose more parallelism, achieving 4.05× speedups of HMult on a GPU, compared to HEAAN on a 24-core CPU.

Jung et al. [29] propose 100x, the first GPU implementation for Bootstrapping CKKS scheme by exploiting massive parallelism available in FHE. The high main-memory bandwidth requirement is regarded as the major performance bottleneck. Based on this observation, they introduce two memory-centric optimizations. (1) Slim Bootstrapping [30] reorders the Bootstrapping process and as such, the computational cost of SlotToCoefficient decreases as the ciphertext level of the input to SlotToCoefficient decreases. (2) Kernel fusion combines multiple GPU kernels together and reuse data in the register file or shared memory, which reduces the number of main-memory accesses between kernels. 100x applies this technique in HE operations such as HMult, HRotate, and HRescale.

Leo de Castro et al. [31] observe that secure implementations of Bootstrapping exhibit a low arithmetic intensity and require large caches. To address the main-memory bandwidth bottleneck, they propose three optimizations. (1) A domain-specific physical address mapping enhances DRAM utilization. (2) The combination and reordering of Algorithms reduce memory and compute requirements of FHE operations. (3) An optimized, memory-aware cryptosystem parameter set maximizes the throughput in Bootstrapping.

Shen et al. [32] present CARM, the first optimized GPU implementation of word-wise HE schemes in IoT scenario, covering BGV, BFV and CKKS. they combine several previous acceleration optimization methods, focus on HMult, and offer various trade-offs between computational efficiency and memory consumption, achieving higher availability and security.

Fan et al. [23] propose TensorFHE, an FHE acceleration solution based on GPGPU with two main optimizations. (1) They utilize Tensor Core Units (TCUs) to boost the computation of NTT. (2) In order to perform sufficient HE operations in a certain time period rather than reducing the latency of one operation, they introduce operation-level batching to fully utilize the data parallelism in GPGPU.

Kim et al. [33] propose Cheddar, an FHE library for CUDA GPUs, which demonstrates significantly faster performance compared to prior GPU implementations. They improve major FHE operations based on efficient kernel designs using a small word size of 32 bits. (1) Extended on-the-fly twiddle factor generation and merged Montgomery transformation eliminate the computational cost of NTT. (2) The best parameters are chosen as BConv setting based on the profiling result.

Yang et al. [34] propose Phantom, an optimized GPU implementations of BGV, BFV and CKKS. (1) They enhance the hybrid key-switching [35] technique, significantly reducing the computational and memory overhead. (2) They explore several kernel fusing strategies to reuse data, resulting in reduced memory access and I/O latency. In a word, the overall implementation demonstrates strong effectiveness.

Fan et al. propose WarpDrive, a comprehensive framework for GPU-based FHE acceleration. The three main optimizations are as follows. (1) A fine-grained memory access strategy reduces the pipeline stalls. (2) The concurrent usage of CUDA and Tensor cores within NTT improves the utilization of heterogeneous GPU units. (3) A parallelism-enhanced kernel design fully exploits the intra-ciphertext parallelism.

Morshed et al. [36] improve the performance of the TFHE scheme by designing efficient parallel frameworks. (1) They extend the boolean gate operations (AND, XOR) to higher level algebraic circuits (addition, multiplications). (2) They devise boolean gates using the GPU parallelism, and employ novel optimizations such as bit coalescing, compound gates, and tree-based additions to implement the higher level algebraic circuits. (3) They modify and incorporate parallelism to the existing algorithms: Karatsuba [37] and Cannon for multiplication and matrix counterpart to achieve further speedup.

TABLE III summarizes the execution time of several GPU-based accelerators for full CKKS workloads, while TABLE IV and TABLE V summarize the throughput for TFHE PBS. Due to the lack of key operation implementation in the computing cores, GPUs are not optimized for specific FHE operations such as ModMult. As a result, they are not able to provide the same level of performance as other domain-specific architectures such as FPGA and ASIC. Besides, large power consumption is the major weakness of GPU.

B. *FPGA-based accelerators*

FPGAs provide a flexible platform for customizing tasks, offering low-level parallelism ideal for accelerating polynomial arithmetic and modular reductions in FHE schemes. They allow hardware tailored to specific encryption operations, delivering significant performance gains over CPUs and GPUs at the cost of difficult design.

As a customizable-computing device, FHE accelerators based on FPGA primarily focus on the efficient parallel computing implementation of primitive-level operations such as NTT, INTT and ModMult [18], [38]–[44]. Fast encryption and decryption operations are also proposed in the literature [45]. Moreover, the data access patterns and data distribution strategies are the key points to consider.

Riazi et al. [46] propose HEAX, the first accelerator for CKKS FHE scheme based on FPGA. (1) They design a new highly parallelizable structure for NTT. (2) Building on top of the NTT engine, they design a novel architecture for computation on homomorphically encrypted data. They evaluate HEAX on a wide range of FHE parameters, achieving 200× overall performance speedup over CPU.

Kim et al. [43] seriously consider the critical Bootstrapping that allows HE operations on encrypted data. (1) They suggest bootstrappable parameter sets for an RNS based HE algorithm. (2) They propose a highly pipelined NTT architecture that uses various levels of parallelism between coefficients, NTT stages and moduli. (3) They reduce the number of roots of unity stored in block RAMs (BRAMs) from $O(N)$ to $O(logN)$ and generate others on the fly.

Han et al. [47] propose an FPGA accelerator design framework for CKKS-based HE. The greatest contribution lies in that they remove data dependencies within KeySwitching and propose a low latency design of KeySwitching module, reducing its latency by up to 48%.

Since FPGA resources and data movement bandwidth are limited, Yang et al. [10] propose Poseidon, a practical FPGA-based FHE accelerator. They provide an FHE-specific microarchitectural design paradigm based on operator reuse, and these operators are combined and reused to implement higher-level FHE operations. In particular, they implement a hardware-friendly automorphism operator called HFAuto, which converts the data movement on the super-long vector into the swap operation of the sub-vectors in an arbitrary granularity.

Agrawal et al. [48] propose FAB, a novel accelerator that supports all HE operations, including fully-packed Bootstrapping, in the CKKS scheme for practical FHE parameters.

(1) They identify an optimal FHE parameter set that can support CKKS Bootstrapping as well as the computational requirement of a real-time machine learning application. (2) They address the memory limitations of Bootstrappable FHE by employing smart operation scheduling and on-chip memory management techniques, thus maximizing the overall computing throughput. In short, FAB achieves performance that is competitive with ASIC, exhibits accessibility of a pure CPU/GPU implementation and demonstrates FPGAs to be a sweet-spot for Bootstrappable FHE.

Zhu et al. [49] propose FxHENN, the first full-fledged FPGA acceleration framework for FHE-based convolution neural network (HE-CNN) inference. They design parameterized HE operation modules with intra- and inter- HE-CNN layer resource management based on FPGA high-level synthesis (HLS) design flow. With sophisticated resource and performance modeling of the HE operation modules, the proposed FxHENN framework automatically performs design space exploration to determine the optimized resource provisioning and generates the accelerator circuit for a given HE-CNN model on a target FPGA device.

Ren et al. [50] propose CHAM, a customized accelerator for high-performance homomorphic matrix vector product (HMVP) with algorithm-architecture co-design. (1) CHAM supports different types of ciphertexts (RLWE and LWE) and the conversion between them, which demonstrates more flexibility than conventional algorithms. (2) CHAM features a novel, compute-efficient NTT architecture that enables pipelined data flow without any bubble. Based on the above optimizations, CHAM is the first HE accelerator deployed for commercial applications.

Bootstrapping remains the main performance bottleneck because it is inherently sequential and exhibits interdependency among the data. Based on this observation, Rashmi Agrawal et al. [51] propose HEAP, an accelerator employing a hybrid scheme-switching approach. (1) HEAP uses the CKKS scheme for the non-bootstrapping steps, but switches to the TFHE scheme when performing the bootstrapping step of the CKKS scheme. (2) HEAP introduces a variety of hardware optimizations in HEAP from modular arithmetic level to NTT and BlindRotate datapath.

Yang et al. propose Hydra, a high-performance FHE acceleration architecture in a scale-out manner for secure deep learning. Hydra supports the multi-server scaling and arbitrary computational nodes theoretically, each handling a portion of the deep learning model governed by the central scheduling mechanism on the host server. Hydra exhibits excellent scalability and delivers outstanding performance across various computing resource scales.

Ye et al. [52] propose an accelerator for TFHE computations based on FPGA. (1) To enable efficient multi-level parallelism, we customize the data layout of TFHE ciphertext for FPGA on chip SRAM to optimize data access and reduce memory access conflict. (2) They parameterize the FPGA design for TFHE Bootstrapping, which can be configured to achieve high performance for different user-specified security requirements and given FPGA resources.

Nam et al. [53] propose XHEC, an accelerator with multiple Tgate processing units (TGCs) to maximize performance of a TFHE N-bit operation. (1) They propose a new scheduling algorithm for maximum utilization ratio of TGCs that enhances the overall performance when XHEC evaluates N-bit ALR operations. (2) They optimize the performance of each TGC by restructuring data layout for minimum transfer latency and operating highly parallelized element-wise processing units.

TABLE III summarizes the execution time of several FPGA-based accelerators for full CKKS workloads, while TABLE IV and TABLE V summarize the throughput for TFHE PBS. In general, FPGAs outperform GPUs in HE computation. However, FPGAs have disadvantages of high development complexity and extended development time. Although FPGAs achieve remarkable acceleration, their performance is not as high as that of ASICs. When handling large-scale computations, the throughput becomes a bottleneck.

TABLE III
EXECUTION TIME FOR FULL CKKS WORKLOADS

Type	Accelerator	Boot (ms) [2]	HELR (ms) [1]	ResNet (s) [3]
GPU	100x [29]	328	775	—
GPU	TensorFHE [23]	250	220	4.94
GPU	GME [54]	33.6	54.5	0.98
GPU	Cheddar [33]	53.4	54.1	1.67
GPU	WarpDrive	97	78	4.77
GPU	Anaheim	29.3	41.2	1.02
FPGA	FAB [48]	477	103	—
FPGA	Poseidon [10]	128	72.9	2.66
FPGA	HEAP [51]	7.8	7	0.27
ASIC	F1 [16]	1024	—	2.69
ASIC	CraterLake [17]	6.33	3.81	0.32
ASIC	BTS [55]	28.6	28.4	1.91
ASIC	ARK [11]	3.52	7.42	0.13
ASIC	SHARP [22]	3.12	5.23	0.10
ASIC	Trinity [56]	1.92	1.37	0.09

C. ASIC-based accelerators

ASICs are custom-designed hardware chips and provide the highest performance and energy efficiency for FHE applications. The overall performance of CKKS accelerators suffers from frequent data transfers due to the partial acceleration support. ASIC-based Accelerators can support all FHE operations and significantly improve the performance of workloads [55]. However, ASICs are expensive to develop and inflexible once fabricated.

Samardzic et al. [16] propose F1, the first ASIC-based FHE accelerator for the BGV scheme that is programmable and capable of executing full FHE programs. F1 is equipped with novel functional units deeply specialized to FHE primitives, such as ModMult, NTT and Automorphism. This has greatly improved the throughput, but the resulting data movement becomes the bottleneck. Therefore, an explicitly managed memory hierarchy and mechanisms decouple data movement from execution and a novel compiler leverages these mechanisms to maximize reuse and schedule off-chip and on-chip data movement. In a word, F1 lays a solid foundation for the subsequent CKKS FHE accelerators.

Based on F1, Samardzic et al. [17] propose CraterLake, the first FHE accelerator that enables FHE programs of unbounded size. Given that F1 targets shallow computations and is tailored to a KeySwitching algorithm that does not scale to high multiplicative budgets L, CraterLake adopts a new, simpler hardware organization and data tiling approach that reduces communication and scales to the large ciphertexts required. and it is tailored to use an efficient KeySwitching algorithm, namely Boosted KeySwitching, which requires new functional units and optimizations. (1) Boosted KeySwitching is dominated by the BConv unit, which exploits the high internal reuse in function changeRNSBase to allow much higher throughput than independent multipliers and adders communicating through the register file. (2) As half of each KeySwitching hint (KSH) is pseudo-random, it can be generated on the fly from a small seed, halving KSH storage and bandwidth, they propose the first hardware KSH Generator. (3) Moreover, CraterLake optimizes F1 in some parameter selections and propose vector chaining to reduce register file port pressure. Based on the above optimizations, CraterLake outperforms F1 by 11.2×.

Kim et al. [55] propose BTS, a bootstrappable, technology driven, secure accelerator for FHE. (1) They provide a detailed analysis of the interplay of HE parameters impacting the performance of FHE accelerators. (2) BTS is equipped with NoCs tailored to the mathematical traits of FHE operations and massively parallel computing primitive units, such as NTT and Automorphism. Based on the above optimizations, BTS shows 5,556× and 1,306× improved execution time on ResNet-20 and logistic regression over CPU.

Kim et al. [11] propose ARK, an accelerator for FHE with runtime data generation and inter-operation key reuse, which enables practical FHE workloads with a novel algorithm-hardware co-design to accelerate Bootstrapping. (1) They first analyze CKKS bootstrapping in detail and identify the memory bottleneck in hardware acceleration of FHE induced by excessive single-use data during bootstrapping. (2) They devise multi-hop HRot to resolve the memory bottleneck of the two memory-intensive computational patterns and on-the-fly limb extension to generate the limbs of plaintexts used in PMult and PAdd on-the-fly to save off-chip memory access. (3) They deploy specialized function units that minimize on-chip data movement, such as BConv, NTT and Automorphism.

Kim et al. [22] propose SHARP, a robust and practical accelerator for FHE. (1) They conduct a comprehensive analysis of the impact of machine word length choice on the FHE acceleration, revealing that a relatively short word length of 36 bits is a robust and efficient solution for FHE accelerators. (2) In order to minimize the memory and communication bandwidth usage, they propose a novel hierarchical organization and functional units specialized for the organization, such as element-wise-engine and ten-step NTT. (3) They devise architectural and algorithmic optimizations that enable the use of significantly smaller on-chip memory while preventing frequent off-chip memory access.

Jiang et al. [57] propose MATCHA, a fast and energy-efficient accelerator to process TFHE gates. MATCHA supports aggressive Bootstrapping key unrolling to accelerate TFHE gates without decryption errors by approximate multiplication-less integer FFTs and IFFTs, and a pipelined datapath.

Putra et al. [26] propose Strix, which utilizes streaming and fully pipelined architecture with specialized functional units to accelerate the sequential ciphertext processing in TFHE. (1) They propose a two-level batching approach to substantially enhance the batch size in PBS. (2) They propose a novel microarchitecture of Decomposition, which is suitable for processing streaming data at high throughput. (3) They utilize a fully-pipelined FFT microarchitecture to eliminate the complex memory access bottleneck and enhance its performance through a folding scheme.

Prasetiyo et al. [58] propose Morphling, an accelerator architecture for the TFHE scheme. (1) Observing that domain-transform operations (FFT) is the main bottleneck, they combine the 2D systolic array and strategic use of transform-domain reuse in order to reduce the overhead of domain-transform. (2) They optimize microarchitecture design for end-to-end TFHE operations such as FFT, Rotation and specialized buffer design. (3) They introduce custom instructions for tiling, batching, and scheduling of multiple ciphertext operations, which facilitates software-hardware co-optimization.

TABLE III summarizes the execution time of several ASIC-based accelerators for full CKKS workloads, while TABLE IV and TABLE V summarize the throughput for TFHE PBS. In spite of the high performance, ASIC-based solutions require significant on-chip storage [16], which leads to overhigh implementation cost.

TABLE IV
PARAMETER SETTING FOR THE TFHE SCHEME

Set	N	n_{lwe}	k	l_b	λ
I	2^{10}	500	1	2	80-bit
II	2^{10}	630	1	3	110-bit
III	2^{11}	592	1	3	128-bit

TABLE V
THROUGHPUT FOR TFHE PBS (OPERATIONS PER SECOND)

Type	Accelerator	TP Set-I	TP Set-II	TP Set-III
CPU	Concrete	63	36	12
GPU	NuFHE	2,500	550	—
FPGA	YKP [52]	2,657	—	836
FPGA	XHEC [53]	2,200	1,800	—
ASIC	MATCHA [57]	10,000	—	—
ASIC	Strix [26]	74,696	39,600	21,104
ASIC	Morphing [58]	147,615	78,692	41,850
ASIC	Trinity [56]	660,060	340,136	180,987

D. Other optimizations

Multi-modal. Homomorphic encryption computation can be classified into arithmetic homomorphism and logical homomorphism based on load characteristics. Currently, HE depends on accelerators to meet the performance requirements of practical applications, which involves different types of scheme. Additionally, using multiple schemes within the same application also involves data conversion between different schemes. Therefore, how to support multiple homomorphic encryption schemes and their conversion algorithms based on a unified hardware architecture becomes an important issue.

Deng et al. [56] propose Trinity, the first multi-modal accelerator based on a unified architecture, which efficiently supports CKKS, TFHE, and their conversion scheme within a single accelerator. (1) They conduct a detailed investigation into the opportunities and challenges of supporting various FHE schemes and their conversion algorithms within the unified architecture. (2) They explore to provide high-performance support for various FHE kernels, such as NTT with different polynomial lengths, NTT and MAC workload with changing proportion. The performance of Trinity outperforms the state-of-the-art accelerator for CKKS (SHARP) and TFHE (Morphling) by 1.49× and 4.23×, respectively.

Zhou et al. propose UFC, a unified accelerator that provides better performance and cost-efficiency than prior scheme-specific accelerators on hybrid FHE applications. (1) They exploit several algorithm-hardware co-design techniques to minimize the hardware cost, which enables high-throughput implementation of function units. (2) They devise several compiler-level optimizations that efficiently utilize the hardware for various FHE operations. (3) They conduct a comprehensive design space exploration on the architectural parameters of UFC, and evaluate FHE workloads in various scenarios, including scheme-specific and hybrid workloads.

Memory. Recent research has increasingly focused on optimizing homomorphic encryption from a memory-centric perspective to address its high computational and storage overhead. Several innovative approaches are being explored. (1) Memory-Aware Design Techniques optimize HE algorithms and hardware co-design, focusing on data locality, cache-efficient scheduling, and tailored memory hierarchies to reduce access latency. (2) Process-in-Memory (PIM) integrates computation directly within memory arrays, leveraging high bandwidth and parallelism to accelerate HE operations while minimizing data movement. (3) Near-DRAM Processing places specialized HE accelerators close to DRAM, mitigating the memory wall problem and improving throughput for large-scale ciphertext operations.

Observing that CKKS bootstrapping exhibits a low arithmetic intensity, Agrawal et al. [59] propose memory-aware design (MAD) techniques to accelerate the bootstrapping operation of the CKKS FHE scheme, which can be equally applied to GPUs, CPUs, FPGAs, and ASICs. (1) MAD makes use of several caching optimizations that enable maximal data reuse and performs reordering of operations to reduce the amount of data that needs to be transferred to/from the main memory. (2) MAD includes several algorithmic optimizations that reduce the number of data access pattern switches and the expensive NTT operations. (3) Near-DRAM Processing (NDP) places specialized HE accelerators close to DRAM, mitigating the memory wall problem and improving throughput for large-scale ciphertext operations.

Kim et al. propose Anaheim, a PIM for FHE with strong practicality. They discover that the bottleneck of performance on GPUs primarily lies in simpler element-wise operations, which are limited by off-chip memory (DRAM) bandwidth. Based on this observation, they make three optimizations. (1) They develop an end-to-end software framework for using PIM with GPUs to optimize FHE execution flows. (2) They design a versatile PIM unit to handle various modular integer arithmetic PIM instructions. (3) They introduce an efficient data mapping and associated PIM execution algorithms to minimize data access overhead.

Park et al. propose FHENDI, a hierarchical NDP solution for FHE applications that harnesses the massive DRAM bank bandwidth. (1) Due to the lack of bank-to-bank communication support, they propose a novel NDP interconnect structure based on the Singleton FFT algorithm [60] and propose a conflict-free mapping algorithm to efficiently execute expensive FHE primitives, such as NTT and Automorphism. (2) For limited die-to-die bandwidth, they devise a scheme to efficiently execute parallel Bootstrapping operations by pipelining its execution across multiple dies. (3) To hide the large memory access latency, they exploit a dual-banking scheme and subarray-level parallelism (SLP) of the DRAM banks.

Compile. Recent research has started to focus on achieving end-to-end acceleration of homomorphic encryption at the compiler level, aiming to enhance the efficiency of encryption and decryption operations through automated optimization techniques. These studies not only focus on traditional operator optimization but also involve deep optimization of encryption-specific operations, such as data partitioning and parallel processing. By combining efficient memory management, hardware acceleration, and intelligent algorithm selection, end-to-end optimization can significantly reduce the computational time of homomorphic encryption in practical applications, promoting its widespread use in fields like data privacy protection and cloud computing.

Jiang et al. propose FHE-CGRA, a coarse-grained reconfigurable architecture (CGRA) acceleration framework with an MLIR-based compiler toolchain for end-to-end homomorphic applications. (1) They demonstrate that runtime reconfigurability is an inherent need to build resource-efficient hardware FHE accelerators. (2) They propose a design of configurable and efficient CGRA primitives with first-order representational support for key FHE dialects, which enables partitioned mapping strategy with high performance and low runtime reconfiguration overhead. (3) They implement an MLIR-based end-to-end compilation framework that can lower an ML model from various ML frontends (ONNX, PyTorch, TensorFlow) to the FHE-dialect. Automatic compiler optimization techniques have been applied to further improve the overall performance of the executable CGRA code.

Huang et al. propose EFFECT, a highly efficient full-stack FHE acceleration platform with a compiler that provides comprehensive optimizations and vector-friendly hardware. (1) To keep high throughput while using extremely small on-chip memory, they propose a novel compiler optimization scheme on the software side with its architectural support. (2) To reduce hardware redundancy without performance penalty, they devise a circuit-level reuse scheme for functions units and specialized source-saving NTT and Automorphism units. They evaluate EFFECT in logistic regression and Bootstrapping with both FPGA and ASIC versions.

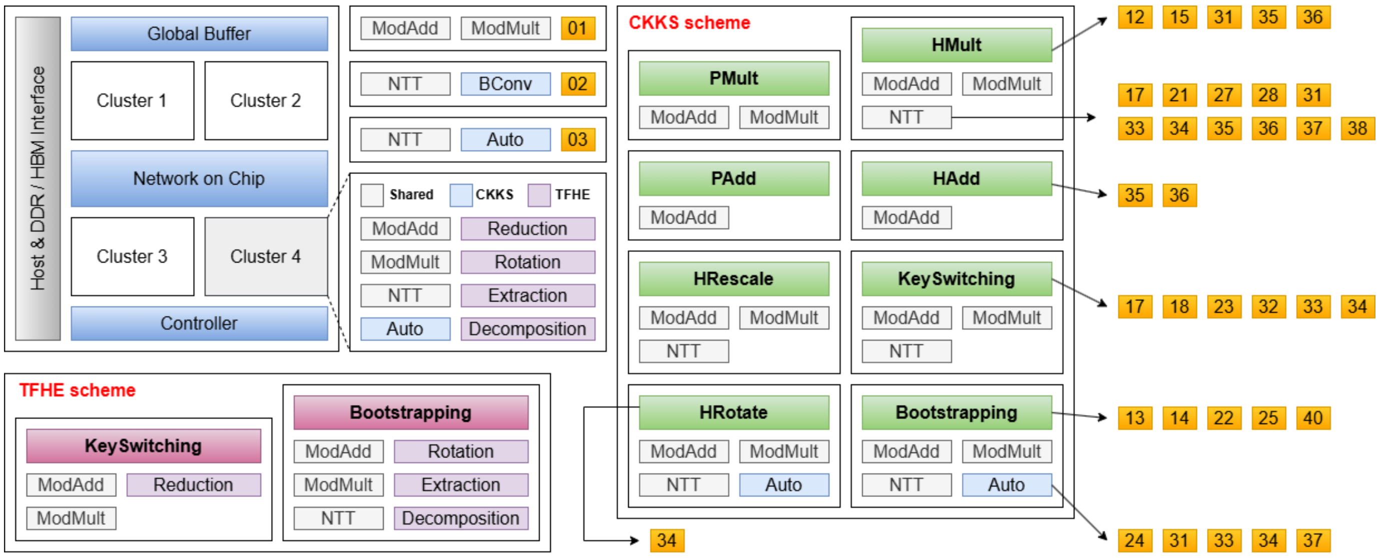

Fig. 5. Key optimizations for different accelerators

V. Inspirations and Future Research Directions

Through the previous discussion, we have gained a clear and in-depth understanding of the current development of hardware accelerators for homomorphic encryption. Figure 5 shows the key optimizations for different accelerators. However, there are still shortcomings and area for improvement. In this section, we list several directions worthy of further investigation to inspire the following research. Specifically, from the perspective of primitive-level operation, algorithm optimization and novel architectures enhances computational efficiency while higher utilization reduces power consumption. Besides, from the perspective of function-level operation, well-designed scheduling is key to improving computational efficiency and reducing storage costs.

A. *Primitive-level operation*

Advanced algorithm. Algorithmic optimizations are the fundamental of improving FHE computational efficiency. For example, the proposal of RNS reduces the high complexity of performing multi-precision arithmetic between large integer coefficients. Besides, the successive introduction and application of Karatsuba, Montgomery, and Barrett multiplication techniques greatly facilitates the computation of ModMult. In the future, more advanced algorithms are worth exploring to reduce time complexity.

Novel hardware architecture. A number of novel hardware architecture optimizations with regard to specific primitive-level operations have also been proposed, such as four-step NTT [16], ten-step NTT [22], on-the-fly twisting factor generation [25]. In specific, compared to four-step NTT, ten-step NTT reduces the high communication overhead while on-the-fly twisting factor generation is beneficial for memory management. In the future, more novel hardware architectures are worth exploring to reduce communication overhead and storage costs.

Higher utilization. Improving hardware utilization is crucial for reducing energy consumption. Based on the above investigation, we summarize the following methods.

(1) Hardware normalization. Primitive-level operations other than NTT, BConv and Automorphism are all element-wise operations. Kim et al. [22] (SHARP) propose a versatile element-wise engine (EWE) that supports five instructions, covering all compound element-wise computation patterns for the CKKS scheme. Input operands are optional to make each instruction serve multiple purposes. Inspired by this, a centralized structure covering several computation patterns both for the CKKS and TFHE scheme is worth exploring to minimize the hardware and software complexity.

(2) Hardware reuse. Zhou et al. (UFC) exploit the basic theory of NTT and reuse NTT interconnect to process Automorphism by setting the appropriate root of the unit for computation, in spite of extra NTT and INTT computation. Besides, while FFT is commonly used in the TFHE scheme to accelerate External Product, it is possible to substitute FFT with NTT by selecting a prime modulus p, which satisfies $p \equiv 1 \bmod 2N$ and is chosen to be the closest prime to q [52], [61]. Hardware reuse plays an important role in FHE computation, especially for the hybrid scheme. Therefore, exploring more reuse opportunities in hardware and finding corresponding methods to mitigate the additional overhead it brings are key strategies to enhance hardware utilization.

(3) Hardware heterogeneous. Deng et al. [56] (Trinity) creatively propose the processing element (PE) structure, providing different compute patterns for NTT and MAC operations, which greatly improves the hardware utilization. Through ALU-level custom design, the energy efficiency of FHE schemes in hardware implementation can be further optimized. Hardware heterogeneous allows different computing units to dynamically adjust based on workload demands, which not only improves performance but also effectively reduces power consumption.

B. Function-level operation

Well-designed Scheduling. During executing function-level operations, prior works have attempted to expose more parallelism by reordering different primitive-level operations. For example, Jung et al. [29] adopt slim Bootstrapping [30] and reorder the Bootstrapping process to reduce the computational cost. Leo de Castro et al. [31] make reordering optimizations to the ModDown operation, thus saving DRAM transfers. Jung et al. [28] employ loop reordering to expose more parallelism in HMult, achieving significant acceleration. Therefore, it is important for function-level operations to arrange the execution order of the decomposed primitive-level operations and implement a highly pipelined structure.

C. Practical application

Parameter selection for specific applications. Another research direction is developing methods to choose the most suitable parameters (such as polynomial degree, modulus, and scaling factor) based on the specific needs of the application. Different applications requires different levels of precision and performance. For example, machine learning models may need parameters that balance accuracy and computational efficiency, while other applications may prioritize faster execution or lower power consumption. Therefore, when FHE is applied to Deep Neural Network (DNN), the corresponding optimal parameter can be selected for each layer of the network. On the other hand, researchers can focus on adaptive techniques where parameters are dynamically adjusted based on the complexity of the data or operations being processed. This will ensure that resources are used efficiently without compromising security or performance.

VI. Conclusion

By conducting an analysis and comparison of several hardware accelerators for fully homomorphic encryption (FHE), especially for the CKKS and TFHE scheme, we explore the future research directions from multiple perspectives. This paper aims to provide readers with a clear, comprehensive and in-depth insight into FHE as well as its acceleration methods, and offer researchers in related field valuable guidance and support, pointing out the potential directions for future research. It is our belief that the novel ideas and practical solutions proposed in this paper is bound to have a positive impact and drive on the research and application of hardware accelerators for FHE, and promote the advancement and dissemination of FHE.

VII. Acknowledgments

We sincerely thank the anonymous reviewers for their insightful and valuable feedback. This work is supported by

References

[1] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 9466–9471.

[2] J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation," in *International Conference on Applied Cryptography and Network Security*. Springer, 2022, pp. 521–541.

[3] R. Podschwadt and D. Takabi, "Classification of encrypted word embeddings using recurrent neural networks." in *PrivateNLP@ WSDM*, 2020, pp. 27–31.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[5] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual cryptology conference*. Springer, 2012, pp. 868–886.

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in cryptology–ASIACRYPT 2017: 23rd international conference on the theory and applications of cryptology and information security, Hong kong, China, December 3-7, 2017, proceedings, part i 23*. Springer, 2017, pp. 409–437.

[7] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.

[8] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[9] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 2019, pp. 347–368.

[10] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 870–881.

[11] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. 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)*. IEEE, 2022, pp. 1237–1254.

[12] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer, 2021, pp. 1–19.

[13] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellán, J. Kim, and D. Kaeli, "Accelerating polynomial multiplication for homomorphic encryption on gpus," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 61–72.

[14] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[15] E. Alkım, Y. A. Bilgin, and M. Cenk, "Compact and simple rlwe based key encapsulation mechanism," in *Progress in Cryptology–LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2–4, 2019, Proceedings 6*. Springer, 2019, pp. 237–256.

[16] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[17] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[18] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2004.

[19] B. Zhang and S. Yan, "Area-efficient barrett modular multiplication with optimized karatsuba algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[20] B. Zhang, Z. Cheng, and M. Pedram, "An iterative montgomery modular multiplication algorithm with low area-time product," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 236–249, 2022.

[21] ——, "Design of a high-performance iterative barrett modular multiplier for crypto systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2024.

[22] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[23] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 922–934.

[24] D. H. Bailey, "Ffts in external or hierarchical memory," *The journal of Supercomputing*, vol. 4, pp. 23–35, 1990.

[25] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 264–275.

[26] A. Putra, Prasetiyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1319–1331.

[27] A. Al Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "Privft: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226 544–226 556, 2020.

[28] W. Jung, E. Lee, S. Kim, J. Kim, N. Kim, K. Lee, C. Min, J. H. Cheon, and J. H. Ahn, "Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *IEEE Access*, vol. 9, pp. 98 772–98 789, 2021.

[29] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[30] H. Chen and K. Han, "Homomorphic lower digits removal and improved fhe bootstrapping," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 315–337.

[31] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does fully homomorphic encryption need compute acceleration?" *arXiv preprint arXiv:2112.06396*, 2021.

[32] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "Carm: Cuda-accelerated rns multiplication in word-wise homomorphic encryption schemes for internet of things," *IEEE Transactions on Computers*, vol. 72, no. 7, pp. 1999–2010, 2022.

[33] J. Kim, W. Choi, and J. H. Ahn, "Cheddar: A swift fully homomorphic encryption library for cuda gpus," *arXiv preprint arXiv:2407.13055*, 2024.

[34] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A cuda-accelerated word-wise homomorphic encryption library," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4895–4906, 2024.

[35] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.

[36] T. Morshed, M. M. Al Aziz, and N. Mohammed, "Cpu and gpu accelerated fully homomorphic encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 142–153.

[37] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[38] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Optimised multiplication architectures for accelerating fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2794–2806, 2015.

[39] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "High-speed fully homomorphic encryption over the integers," in *Financial Cryptography and Data Security: FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers 18*. Springer, 2014, pp. 169–180.

[40] C. M. Moore, "Accelerating fully homomorphic encryption over the integers," Ph.D. dissertation, Queen's University Belfast, 2015.

[41] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *2013 Euromicro conference on digital system design*. IEEE, 2013, pp. 955–962.

[42] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.

[43] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. 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)*. IEEE, 2020, pp. 56–64.

[44] W. Wang, X. Huang, N. Emmart, and C. Weems, "Vlsi design of a large-number multiplier for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 1879–1887, 2013.

[45] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, 2014.

[46] M. S. Riazi, K. Laine, B. Pelton, and W. 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*, 2020, pp. 1295–1309.

[47] M. Han, Y. Zhu, Q. Lou, Z. Zhou, S. Guo, and L. Ju, "coxhe: A software-hardware co-design framework for fpga acceleration of homomorphic computation," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1353–1358.

[48] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International symposium on high-performance computer architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[49] Y. Zhu, X. Wang, L. Ju, and S. Guo, "Fxhenn: Fpga-based acceleration framework for homomorphic encrypted cnn inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 896–907.

[50] X. Ren, Z. Chen, Z. Gu, Y. Lu, R. Zhong, W.-J. Lu, J. Zhang, Y. Zhang, H. Wu, X. Zheng *et al.*, "Cham: A customized homomorphic encryption accelerator for fast matrix-vector product," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[51] R. Agrawal, A. Chandrakasan, and A. Joshi, "Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 756–769.

[52] T. Ye, R. Kannan, and V. K. Prasanna, "Fpga acceleration of fully homomorphic encryption over the torus," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[53] K. Nam, H. Oh, H. Moon, and Y. Paek, "Accelerating n-bit operations over tfhe on commodity cpu-fpga," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[54] K. Shivdikar, Y. Bao, R. Agrawal, M. Shen, G. Jonatan, E. Mora, A. Ingare, N. Livesay, J. L. Abellán, J. Kim *et al.*, "Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 670–684.

[55] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th annual international symposium on computer architecture*, 2022, pp. 711–725.

[56] X. Deng, S. Fan, Z. Hu, Z. Tian, Z. Yang, J. Yu, D. Cao, D. Meng, R. Hou, M. Li *et al.*, "Trinity: A general purpose fhe accelerator," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 338–351.

[57] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 235–240.

[58] A. Putra, J.-Y. Kim *et al.*, "Morphling: A throughput-maximized tfhe-based accelerator using transform-domain reuse," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 249–262.

[59] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 685–697.

[60] R. Singleton, "An algorithm for computing the mixed radix fast fourier transform," *IEEE Transactions on audio and electroacoustics*, vol. 17, no. 2, pp. 93–103, 1969.

[61] M. Joye and M. Walter, “Liberating tfhe: programmable bootstrapping with general quotient polynomials,” in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 1–11.