SoK: Fully Homomorphic Encryption Accelerators

JUNXUE ZHANG*, iSINGLab @ HKUST, Hong Kong, China and Clustar, China
XIAODIAN CHENG*, iSINGLab @ HKUST, Hong Kong, China
LIU YANG, iSINGLab @ HKUST, Hong Kong, China and Clustar, China
JINBIN HU, iSINGLab @ HKUST, Hong Kong, China
XIMENG LIU, Fuzhou University, China
KAI CHEN, iSINGLab @ HKUST, Hong Kong, China

Fully Homomorphic Encryption (FHE) is a key technology enabling privacy-preserving computing. However, the fundamental challenge of FHE is its inefficiency, due primarily to the underlying polynomial computations with high computation complexity and extremely time-consuming ciphertext maintenance operations. To tackle this challenge, various FHE accelerators have recently been proposed by both research and industrial communities. This paper takes the first initiative to conduct a systematic study on the 14 FHE accelerators — cuHE/cuFHE, nuFHE, HEAT, HEAX, HEXL, HEXL-FPGA, 100×, F1, CraterLake, BTS, ARK, Poseidon, FAB and TensorFHE. We first make our observations on the evolution trajectory of these existing FHE accelerators to establish a qualitative connection between them. Then, we perform testbed evaluations of representative open-source FHE accelerators to provide a quantitative comparison on them. Finally, with the insights learned from both qualitative and quantitative studies, we discuss potential directions to inform the future design and implementation for FHE accelerators.

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Security and privacy → Cryptography**.

Additional Key Words and Phrases: Fully Homomorphic Encryption, Accelerator

1 Introduction

With the increasing concern about data privacy and integrity, privacy-preserving computing has been adopted in many real-world applications, *e.g.*, cloud computing [84], machine learning [73, 87], database search [74], *etc.* Among all the privacy-preserving technologies, fully homomorphic encryption (FHE) emerges as one of the most important and promising technologies and has been adopted in various applications [35, 67, 74, 88]. Specifically, FHE allows performing arbitrary operations directly over the encrypted data without decryption, making it appealing for privacy-preserving computation.

Although promising, one fundamental drawback of FHE is its *inefficiency*. Compared to plaintext computation, FHE-enabled computation is orders of magnitude slower, which restricts its deployment in many performance-critical systems. To solve this problem, various optimizations have been proposed. One direction is to improve the efficiency of the algorithm. For example, modern FHE schemes, such as BGV [32], BFV [50], and CKKS [37], all support SIMD-like (*i.e.*, batching) operations [83] to pack many plaintext messages into one ciphertext to improve the execution efficiency. Instead of studying these algorithm-level performance optimizations, in this paper we focus on the other key direction — leveraging hardware accelerators to improve the efficiency of FHE schemes[1].

*Both authors contributed equally to this research.
[1]Note that algorithm optimization and hardware acceleration are complementary, and can be combined to improve the overall performance.

Authors' addresses: Junxue Zhang, zjx@cse.ust.hk, iSINGLab @ HKUST, Hong Kong, China and Clustar, China; Xiaodian Cheng, xchengaq@connect.ust.hk, iSINGLab @ HKUST, Hong Kong, China; Liu Yang, lyangau@connect.ust.hk, iSINGLab @ HKUST, Hong Kong, China and Clustar, China; Jinbin Hu, jinbinhu@ust.hk, iSINGLab @ HKUST, Hong Kong, China; Ximeng Liu, snbnix@gmail.com, Fuzhou University, China; Kai Chen, kaichen@cse.ust.hk, iSINGLab @ HKUST, Hong Kong, China.

Before introducing FHE accelerators, we first illustrate what makes FHE slow and the challenges of acceleration. In this paper, we find that the root cause of FHE's inefficiency is two-fold: underlying polynomial computations with high computation complexity and two extremely time-consuming ciphertext maintenance operations. First, most of FHE's underlying operations are polynomial operations, which are much more complex than plaintext computation, where operands are integers or floating numbers. While Fast Fourier Transform (FFT)/Number Theoretic Transform (NTT) can be utilized to speed up the polynomial operations [41] from algorithm-level, further accelerating NTT/FFT faces challenges in three aspects: high computation complexity, extremely intensive memory access, and limited generality (§4.1). Second, compared to plaintext computation, FHE requires ciphertext maintenance operations to ensure correctness. Such operations involve over-complicated computation steps, causing further performance degradation. Although they are built upon polynomial operations, fully accelerating ciphertext maintenance operations is more challenging in terms of the aforementioned three aspects compared to merely accelerating polynomial operations (§4.2 and §4.3).

To improve the efficiency of FHE, FHE accelerators are proposed. Initially, these accelerators rely on features provided by general hardware. For example, Intel proposed Intel Homomorphic Encryption Acceleration Library (HEXL) to leverage AVX-512 instructions for fast NTT operations [28]. nuFHE [5] and 100× [62] used GPU implementations, *i.e.*, CUDA [13] programs, to accelerate TFHE [39] and CKKS [37] respectively. While providing notable acceleration for FHE schemes, these accelerators are far from satisfactory.

To further improve the performance of FHE schemes, people begin to exploit specific-designed hardware accelerators. Field Programmable Gate Array (FPGA) is first used to build circuits to efficiently execute NTT, inverse NTT (iNTT), and key-switching in FHE schemes [11, 42, 43, 47, 71, 72, 78]. These FPGA-based accelerators are affordable but suffer from intrinsic disadvantages of FPGA itself: limited programmable resources and low working frequency [66]. Then, to overcome these disadvantages, people are seeking expensive Application-specific Integrated Circuit (ASIC) technologies to build high-performant FHE accelerators [63, 65, 81, 82]. While the acceleration ratios of these ASIC-based accelerators are promising, *e.g.*, $\sim$ 1000× and $\sim$ 100× NTT throughput compared to FPGA-based and GPU-based solutions respectively, they are way more expensive. For instance, developing and taping out a 12nm ASIC like [82] requires millions of US dollars.

In the recent decade, we have seen an explosive growth of FHE accelerators [11, 42, 43, 47, 63, 65, 71, 72, 78, 80–82], and expect an increasingly more active development of FHE accelerators in the near future. However, we lack a comprehensive and systematic study to shed light on the status quo of existing FHE accelerators, which could inspire the future design and implementation of FHE accelerators.

Motivated by this, we take the first initiative to perform the systematization of knowledge on FHE accelerators. We first review 14 existing FHE accelerators and make observations on the evolution trajectory of these works, which establishes a qualitative connection among them (§5). Then, to give readers a clear view of how these accelerators perform, we present a quantitative analysis of them. Specifically, we use testbed experiments to evaluate the performance of some representative open-source accelerators, and further include the statistics from papers of other well-known but closed-source accelerators for thoroughness (§6). Finally, based on our qualitative and quantitative analysis, we discuss the potential future directions, such as new design tradeoffs, software/hardware co-designs, scaling methodologies, *etc.*, to inform the future design and implementation of FHE accelerators (§7).

Along with the paper, we provide a Docker image[2] that includes all configurations and scripts for performance evaluations of all open-sourced FHE accelerators used in our paper, which can be readily reused by the community.

[2] https://hub.docker.com/r/hpfilter/sok-fhe-accelerator

Related Works: Systematization of knowledge on FHE library [22] and compiler [86] have already been proposed. To the best of our knowledge, our paper is the first systematization of knowledge on FHE accelerators. FHE accelerators are closely related to FHE libraries and compilers. For example, some FHE accelerators are designed to work with particular FHE libraries, *e.g.*, HEAX chooses SEAL [17] as its target library to accelerate. Moreover, an increasing number of FHE accelerators use compilers for a software/hardware co-design [63, 65, 81, 82]. However, few FHE accelerators consider leveraging existing FHE compilers, such as EVA [46], E3 [38], *etc.*, which, in our opinion, still leaves dramatic design space for better performance and flexibility.

2 Preliminaries

In Table 1, we list the notations used in the paper. Here, we define a polynomial A as follows:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \tag{1}$$

The degree of a polynomial is the highest power of the variable x with a non-zero coefficient. Integer n is defined as the degree-bound of the polynomial, which is strictly larger than the degree of the polynomial [41]. In many previous works, n is also called the degree of a polynomial for simplicity. In this paper, we use the term degree rather than degree-bound to refer to n.

2.1 Ciphertexts and Keys

In this paper, all the cryptosystems are asymmetric, which means we have encrypt plaintext m with public key $\mathbf{pk}$ and decrypt ciphertext $\mathbf{ct}$ with secret key s. All the ciphertexts and keys in the algorithms covered in the paper are in the form of polynomials. Therefore, the underlying operations of FHE schemes are all polynomial operations.

2.2 Polynomial Operations

As discussed, polynomial operations, including polynomial additions and multiplications, are the basic building blocks of the FHE algorithms. To implement the operations, there are two common representations of a polynomial: coefficient representation and point-value representation. The polynomials in FHE are naturally stored in the coefficient representation. However, the time complexity of multiplication between polynomials in the coefficient representation is $O(n^2)$, while it can be reduced to $O(n)$ with the point-value representation. A popular approach for representation conversion is the Fast Fourier Transform, which leverages the idea of divide-and-conquer to reduce the conversion time complexity to $O(n \log n)$. Note that FFT only works on complex numbers. For RLWE-based FHE schemes, we also need Number Theoretic Transform, which is a generalization of FFT but works over finite fields. Readers may refer to Appendix A for more details.

2.3 RNS Decomposition

FHE schemes are mainly constructed over polynomial rings, which means we perform polynomial operations with the coefficients modulus Q and P. Q and P are large integers chosen for ciphertext calculation and key generation, respectively. However, polynomial operations with large integers lead to performance degradation [37]. For performance optimization, a common strategy is to use Residue Number System (RNS). RNS decomposes modulus Q and P into the product of several smaller coprime moduli $q_0 \cdot q_1 \cdot \cdots \cdot q_L$ and $p_0 \cdot p_1 \cdot \cdots \cdot p_\alpha$, allowed by the Chinese Remainder Theorem (CRT) [52]. Multiple polynomials with smaller moduli are used to replace the original polynomial. By applying

Notation	Definition
n	Degree of a polynomial.
m	Plaintext.
ct	Ciphertext.
s	Secret key.
pk	Public key.
Q	Coefficient modulus of ciphertext polynomial.
P	Special modulus for the keys.
$\{q_i, i \in [0, L]\}$	A set of moduli. $Q = \prod_{i=0}^{L} q_i$.
$\{p_i, i \in [0, \alpha - 1]\}$	A set of special moduli. $P = \prod_{i=0}^{\alpha-1} p_i$.
L	Multiplicative depth of a fresh ciphertext.
l	Current multiplicative depth of a ciphertext.
dnum	Decomposition number in key-switching.
α	# of special moduli p_i. $\alpha = \lfloor (L+1)/\text{dnum} \rfloor$.
$\{Q_j, j \in [0, \text{dnum}]\}$	A set of modulus factors. $Q_j = \prod_{i=j\alpha}^{(j+1)\alpha-1} q_i$.
swk	Switching key for key-switching.
q	Coefficient modulus of plaintext polynomial.
Δ	Scaling factor.

Table 1. Notations used in this paper.

RNS to FHE, the coefficient size of polynomials is greatly reduced at the cost of decomposing each polynomial into multiple ones. Since the calculation complexity of modular multiplication is roughly proportional to the square of the coefficients' bit width, the overall complexity is reduced. In word-wise FHE algorithm, the value L is also called the multiplicative depth of a ciphertext as the moduli $q_0 \cdot q_1 \cdot \cdots \cdot q_L$ are gradually consumed by the homomorphic multiplications and L usually determines the maximum number of multiplications that can be supported by the FHE ciphertexts.

3 Fully Homomorphic Encryption

Homomorphic encryption is an encryption scheme that allows performing arbitrary computation over ciphertexts without decrypting them. For example, Paillier is an additive homomorphic encryption scheme thus we can perform additions over ciphertext [76], while RSA [79] is a multiplicative homomorphic encryption scheme. In this paper, we are focusing on fully homomorphic encryption (FHE) schemes[3], which allow both additive and multiplicative homomorphic operations.

Current FHE schemes can be categorized into word-wise FHE and bit-wise FHE [69]. Word-wise FHE supports algebraic operations on word-based (or message-based) encrypted data. Moreover, word-wise FHE supports efficient single-instruction-multiple-data (SIMD) style homomorphic operations, *e.g.*, performing homomorphic addition and multiplication over batched plaintext [83]. However, word-wise FHE is not suitable for evaluating non-polynomial functions, *e.g.*, sigmoid/relu functions, which are commonly used in machine learning applications. Examples of word-wise FHE are CKKS [37], BFV [50], BGV [32], *etc.* In contrast, bit-wise FHE supports operations of boolean circuits. As its name indicates, bit-wise FHE schemes encrypt each bit of the plaintext and are usually used to evaluate

[3]Similar to [86], FHE schemes in our paper include leveled FHE.

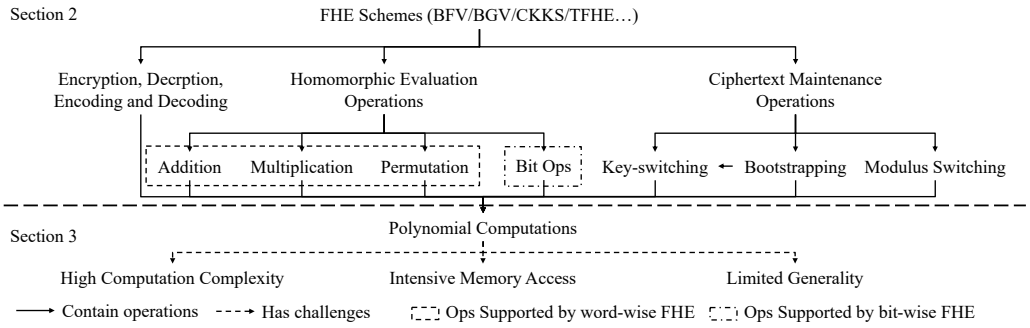

Fig. 1. Overview of the operations used in FHE schemes.

non-polynomial operations by constructing lookup tables. Bit-wise FHE schemes do not provide sufficient support for SIMD-style operations and usually suffer from a higher degree of ciphertext inflation, which poses a larger challenge to memory bandwidth. Examples of bit-wise FHE schemes are TFHE [39] and FHEW [49]. In order to support real-world privacy-preserving applications, such as privacy-preserving machine learning, both word-wise and bit-wise FHE schemes are used together [31, 69].

In this article, we focus on word-wise FHE algorithms constructed over ring learning with errors problem (RLWE) (*e.g.*, BFV, BGV and CKKS) [70] and bit-wise FHE algorithm (*e.g.*, TFHE) [55] because they are practical and widely adopted. Since these FHE schemes mainly manipulate polynomials over finite fields or torus, most of the operations to be discussed later are made up of polynomial computations. In the following sections, we will show them in detail. Figure 1 demonstrates the relationship of operations of FHE.

3.1 Encoding, Decoding, Encryption & Decryption

Encoding and Decoding: The goal of encoding is to convert plaintext messages into polynomials for the subsequent homomorphic operations. Please note that by packing a vector of numbers during encoding, specific FHE schemes, such as BGV and BFV, can naturally support SIMD-style operations. Conversely, decoding is used to recover plaintext messages from polynomials.

Encryption: In an asymmetric encryption system, a ciphertext **ct** is generated by obfuscating the plaintext with a public key and noises. For example, in RLWE-based FHE, the public key is an RLWE instance, $\mathbf{pk} = (\mathrm{pk}_0, \mathrm{pk}_1) = (as + e, a)$, generated from the secret key s. The coefficients of random polynomial a are uniformly sampled from interval $(-Q/2, Q/2]$, and coefficients of noise polynomial e are independently sampled from a discrete Gaussian distribution. The encryption is formulated as

$$\mathbf{ct} = (c_0, c_1) = v \cdot \mathbf{pk} + (\Delta \cdot m + e_1, e_0) \mod Q, \tag{2}$$

where m is the plaintext (*i.e.*, encoding result), e_0 and e_1 are also noise polynomials, v is a random polynomial with small coefficients, and Δ is the scaling factor in some schemes to control precision.

Decryption: Although different FHE schemes use different decryption workflows, the common objective is to recover the plaintext by removing $a \cdot s$ from the ciphertext. As a result, they share the same core operation in decryption, which is $c_0 - c_1 \cdot s$.

3.2 Homomorphic Evaluation

For word-wise FHE, major homomorphic evaluations include multiplications, additions, subtractions and permutations, which are sufficient for most applications. Non-polynomial functions can also be achieved via polynomial approximation, which covers a large range of applications. Bit-wise FHE schemes can efficiently evaluate bit operations, including NOT, AND, NAND, OR and XOR, which can provide accurate results of non-polynomial functions.

However, homomorphic evaluations lead to two significant problems. First, operations such as multiplication and permutation construct special ciphertexts that cannot be directly decrypted or used as the input of subsequent operations. Second, as noise is introduced to secure the ciphertext in FHE schemes, it gradually grows during FHE operations, especially in homomorphic multiplications. After the noise exceeds a threshold, it will impact the correctness of the decryption. Therefore, ciphertext maintenance operations, *e.g.*, key-switching, modulus-switching and bootstrapping, are required in FHE to solve these problems.

3.3 Ciphertext Maintenance

Key-switching: As its name implies, key-switching homomorphically switches the secret key of a ciphertext while keeping the corresponding plaintext unchanged. More specifically, $\mathbf{ct}$ is the ciphertext of plaintext m, and it can be decrypted with some special secret key s'. After executing $\mathbf{ct'} = \text{Keyswitch}(\mathbf{ct}, \mathbf{swk})$, ciphertext $\mathbf{ct'}$ can be decrypted with the original secret key s and the corresponding plaintext is still m. In the equation, $\mathbf{swk}$ is a pre-generated public key called switching key, and it can be considered a ciphertext of $P \cdot s'$ with modulus $P \cdot Q$. P is an integer used to control the scale of noise in key-switching. After certain operations such as homomorphic multiplication and permutation, key-switching is leveraged to convert the resulting ciphertexts back into the original form for the following operations. Therefore, it is intensively used in FHE schemes.

Modulus Switching and Multiplicative Depth: Generally speaking, modulus switching refers to the operations that switch the modulus of a ciphertext. It is largely applied in FHE schemes to raise or reduce the modulus for different purposes. In particular, RLWE-based FHE schemes introduce modulus switching to control the proportion of noise in the ciphertext at the cost of reducing the modulus Q. When Q is too small to support further operations, the noise cannot be reduced anymore. Consequently, the size of modulus Q limits the number of consecutive homomorphic operations on a freshly encrypted ciphertext. The number is also called the maximum multiplicative depth (or budget) L of arithmetics supported by the FHE scheme. Such FHE schemes are usually called leveled fully homomorphic encryption (leveled FHE) because of the limitations on L. In this paper, we use the term deep and shallow to describe applications that consume large multiplicative depth (*e.g.*, deep neural networks [67]) and those that only contain a few multiplications (*e.g.*, database lookup [8]), respectively. Leveled FHE schemes are designed to be efficient for shallow computations. In deep applications, excessive Q dramatically reduces the performance. Bootstrapping is leveraged to refresh the ciphertext and recover the multiplicative depth, which we will discuss next.

Bootstrapping: Bootstrapping is a generic term for operations that refresh a ciphertext in FHE. Their common idea is to homomorphically recrypt the old ciphertext and generate a fresh one.

In bit-wise FHE schemes such as TFHE, bootstrapping is performed in every bit-wise operation. Therefore, multiplicative depth is not considered in TFHE. The most time-consuming part of bootstrapping in TFHE is the homomorphic evaluation of a lookup table (LUT). The lookup process can be implemented through a large number of multiplexer (MUX) gates. Since the MUX gate mainly consists of polynomial additions, subtractions and multiplications, polynomial computations are the major workload in TFHE.

Unlike bit-wise FHE, bootstrapping in RLWE-based word-wise FHE schemes is more complicated. We tend to reduce the frequency of bootstrapping and perform one only when the current multiplicative depth is not enough. It is worth noting that bootstrapping combines multiple operations, including many multiplications, thus consuming considerable multiplicative depth by itself. The concrete implementations of bootstrapping are not the same in various word-wise FHE algorithms, but their workflow can be summarized into four common steps.

Step 1. Modulus Switching: In word-wise FHE, the multiplicative depth is proportional to the modulus size of the ciphertext. Therefore, the modulus should be raised if more multiplications are needed. Given ciphertext ct under modulus Q, the first step in bootstrapping is to generate a new ciphertext ct$'$ encrypting the same plaintext while extending the modulus to Q' which satisfies $Q' \gg Q$.

Step 2. Coeff ToSlot: Although the modulus of the ciphertext has been raised, the decryption formula no longer holds since the coefficients of the plaintext polynomial are not guaranteed to be bounded by the modulus of the plaintext during the process of modulus switching. In this case, homomorphic evaluations are required to modulo the coefficients. However, we can only operate on plaintext slots rather than polynomial coefficients with homomorphic operations in FHE. To make the coefficients accessible for homomorphic evaluations, homomorphic encoding should be performed in advance to put the coefficients of plaintext polynomial into the plaintext slots. This process is also called Coeff ToSlot [36, 60] or linear transformation [34, 58].

Step 3. Homomorphic Evaluation: Homomorphic evaluations in bootstrapping are introduced to modulo the coefficients of plaintext polynomial. They contain non-polynomial functions that can not be directly executed in FHE. For example, in BGV and BFV, the main operation of this step is digit extraction, while modulus reduction is the dominant operation of this step in CKKS. Thus, in modern FHE implementations, approximation schemes such as Taylor expansion [36] and optimized Chebyshev method [34, 60] are used to alternatively evaluate high-degree polynomials.

Step 4. SlotToCoeff: After homomorphic evaluation, the results in the plaintext slots should be placed back to the coefficients of the plaintext polynomial. The process is an inverse operation of Coeff ToSlot, which is called SlotToCoeff or inverse linear transformation.

4 What Makes FHE Slow & the Challenges of Accelerating Them

FHE is an ideal solution for many privacy-preserving applications since it can simultaneously protect confidential data and satisfy emerging data protection lawsuits and regulations [6]. However, FHE still suffers from inefficiency, which is the focus of the paper. It is worth noting that previous works have also mentioned several other reasons restricting the broad adoption of FHE, such as its usage complexity [86]. However, we believe the inefficiency problem is still the major roadblock to FHE's adoption in the production environment.

In this paper, we have identified the cause of FHE's inefficiency as a two-fold problem. First, most of FHE's operations are built upon polynomials [32, 37, 39, 50]. Therefore, FHE is inefficient since polynomial computations, specifically polynomial multiplication, are naturally much more complex than integer/floating number calculations. Second, two necessary ciphertext maintenance operations, *i.e.*, key-switching and bootstrapping, are extremely time-consuming since they involve very complicated computations. As reported in ARK [63], the two major components (*i.e.*, NTT and fast basic conversion, which will be introduced in detail later) in key-switching take up more than 80% of the total computation time. According to the analysis in [82], bootstrapping may take up over 90% of computation time in an end-to-end FHE task (bootstrapping consists of key-switching operations).

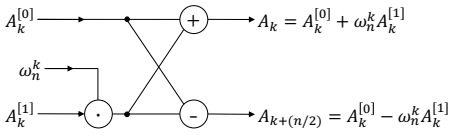

Fig. 2. Cooley-Tukey butterfly

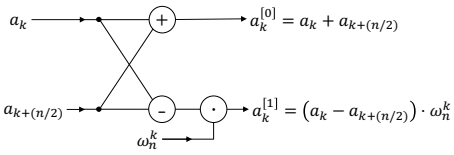

Fig. 3. Gentlemen-Sande butterfly

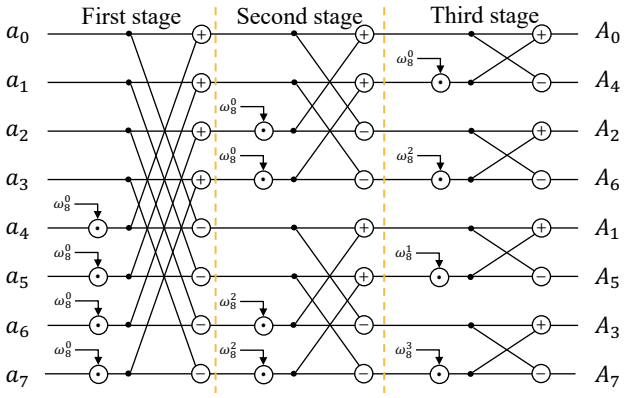

Fig. 4. Workflow of NTT/FFT with $n = 8$ with CT butterfly

To solve the problems, FHE accelerators are proposed. They target improving the performance of FHE schemes by either leveraging general hardware (SIMD feature provided by CPUs) or relying on completely specific hardware (FPGA-based circuits). However, it is not easy for these accelerators to achieve ideal performance. In the following part, we will first summarize the challenges of accelerating polynomial computation (specifically, its core operation: NTT/FFT) into three aspects: *high computation complexity, extremely intensive memory access and limited generality*. Second, we will further demonstrate the challenges of accelerating key-switching and bootstrapping. Although these two operations are built upon polynomials operations, designing an end-to-end acceleration solution for them is more challenging in terms of the aforementioned three aspects, which makes it increasingly more difficult to design efficient FHE accelerators.

4.1 Challenges of Accelerating Polynomial Computation

One widely-adopted algorithm-level optimization for polynomial computation is to leverage Fast Fourier Transform (FFT)/Number Theoretic Transform (NTT) [17, 19]. Thus, accelerating FFT/NTT is essential to all FHE accelerators. In the following part, we will present the three challenges of further accelerating FFT/NTT with the current hardware architecture.

4.1.1 High Computation Complexity Although FFT/NTT has lowered the computation complexity of polynomial multiplications to $O(n \log n)$, this operation still takes the majority of the time, reaching 70% in some scenarios [63]. Therefore, to further accelerate the FFT/NTT, people begin to design accelerators to further improve performance.

Most FFT/NTT implementations are constructed by iteratively executing a basic operation called butterfly. The most well-known butterfly strategies are Cooley-Tukey (CT) butterfly [40] (shown in Figure 2) and Gentlemen-Sande (GS) butterfly [53] (shown in Figure 3). These two strategies mainly consist of addition, subtraction and multiplication operations. CT butterfly is designed to execute the Equation 8 and 10 in Appendix A by reusing the multiplication result. GS butterfly exploits a similar structure and can be used as the reverse operation of CT butterfly in applications.

Based on the butterfly strategies, the FFT/NTT algorithm is divided into $\log n$ stages with $n/2$ butterflies in each stage, as shown in Figure 4, which presents an example of FFT with $n = 8$. The input of the butterflies in FFT/NTT is the output of specific ones in the previous stage, which leads to a strict dependency chain from the first to the last stage. Data dependency among butterflies in adjacent stages continuously changes as the calculation proceeds.

To accelerate FFT/NTT computation, the first attempt is to leverage data parallelism since data parallelism is widely supported by general hardware, such as CPU (through SIMD instructions) and GPU (through CUDA programming). However, data parallelism is only efficient when there are no dependencies among all pieces of data. Consequently, the data parallelism cannot be directly applied to FFT/NTT due to the strict data dependencies among adjacent butterfly operations.

Another accelerating approach is to utilize pipeline parallelism, which can be implemented by designing customized hardware circuits via FPGA or ASIC. However, as Figure 4 indicates, to build a pipeline with large n, $(n \log n)/2$ separate butterflies should be implemented, which will consume too many hardware resources, causing either impossible circuits placement with FPGA or extremely expensive costs with ASIC. Moreover, data movement within such a large pipeline is almost impossible with state-of-the-art hardware technologies.

Therefore, naive data and pipeline parallelism are challenging. There are two possible solutions. First, it is possible to make FFT/NTT partly parallelizable based on the 4-step FFT/NTT algorithm [27]. Readers may refer to the Appendix B for a detailed introduction to 4-step FFT/NTT. The second solution is inter-polynomial parallelism, executing operations over multiple polynomials concurrently with sufficient hardware resources. The scheme is called residue-polynomial-level parallelism (rPLP), and we will discuss it in detail in §4.2.

4.1.2 Intensive Memory Access Recent works have observed that overhead in memory access has become an essential bottleneck even if the accelerator is designed with modern hardware, such as FPGA or ASIC [63, 65, 81, 82]. For example, let's consider a chip operating at 1GHz and there're 40960 on-chip modular multiplication units. The chip is connected to the latest HBM3 with a bandwidth of 3TB/s. Assume that the chip is executing CoeffToSlot with bootstrappable parameters ($n = 2^{16}, L = 23, \mathrm{dnum} = 4$). All the calculation units are concurrently working and the bandwidth of the HBM3 is fully utilized. It only takes the chip 0.18ms to finish all the multiplications, while it takes the HBM3 2.1ms (11.7× of computation time) to load the data used in CoeffToSlot [63].

The root cause of the problem is data inflation. Compared to plaintext computation, data size in polynomial operations largely increases for the following reasons.

In FFT/NTT, a group of pre-computed parameters called twiddle factors is mandatory for the calculations, *i.e.*, ω_n^k in Figure 2 and Figure 3. Because the twiddle factors can be reused throughout an entire FHE job, it is usually cached in the on-chip memory (fast but extremely limited) of the accelerator to achieve high performance. For FFT, the number of twiddle factors is less than the number of input coefficients. However, for NTT, one group of twiddle factors is required for each modulus in RNS, which results in data inflation. In addition to the twiddle factors, the temporary results generated by each stage in FFT/NTT also place a high demand on the memory size.

As mentioned in §4.1.1, the 4-step FFT/NTT algorithm makes it possible to process a single FFT or NTT in parallel. Therefore, it is common to leverage this algorithm in recent designs. However, the 4-step FFT/NTT requires more pre-computed parameters than the original FFT/NTT algorithm (shown in Appendix B.2). Although the algorithm greatly reduces the number of twiddle factors, it introduces n additional pre-computed numbers called twisting factors for each RNS modulus [27]. Therefore, the memory space requirement for pre-computed numbers is even more strict after applying the algorithm optimization.

Furthermore, due to the complex data dependency and large data size, read/write requests over the same block of memory from different butterflies are common. Such conflicts cause either large design complexity or severe performance degradation.

4.1.3 Limited Generality Degree n of the polynomials are variant in different scenarios. For example, n is related to security level and multiplicative depth. However, it is challenging for these specific-designed hardware accelerators, such as FPGA and ASIC, to consistently provide optimal acceleration when n varies. The reason is as follows. There are $(n \log n)/2$ butterflies in the entire FFT/NTT pipeline. When n becomes larger, the consumption of hardware resources greatly increases. Furthermore, different n leads to completely different data dependencies in FFT/NTT. Static circuit connections among butterflies cannot satisfy the requirement.

If we use a fixed n to design the pipeline, the design suffers from resource underutilization or non-optimal performance. For example, when an architecture designed for a certain degree is directly applied to accelerate FFT/NTT with a relatively larger degree, the calculation fails to achieve full pipelining because of insufficient multipliers. In contrast, adapting the design to scenarios where the degree is relatively smaller results in a tremendous waste of computing resources in the redundant stages.

4.2 Challenges of Accelerating Key-switching

As mentioned in §3.3, key-switching is widely used for ciphertext conversion in homomorphic multiplication and permutation. The main challenge of accelerating key-switching is achieving generality, which we will demonstrate in detail in the following parts.

The key-switching algorithms used in state-of-the-art accelerators can be considered as different variants of the same algorithm, *i.e.*, generalized key-switching[54, 60]. Based on RNS decomposition, the modulus Q is factorized into $L+1$ coprime moduli q_0, q_1, ..., and q_L. In generalized key-switching, given a fixed integer parameter dnum $\in [1, L+1]$ which stands for decomposition number, the moduli are grouped into dnum blocks and the partial production of moduli in each block is denoted as $Q_j = \prod_{i=j\alpha}^{(j+1)\alpha-1} q_i$, where $j \in [0, \text{dnum})$ and $\alpha = \lfloor (L+1)/\text{dnum} \rfloor$. Then the key-switching can be decomposed into calculations in each block, followed by the accumulation of results from different blocks. Each block's calculations are also called fast basis conversion, mainly consisting of multiplications and additions, while the major operation in accumulation is NTT. The selection of dnum greatly impacts the memory requirement and calculation complexity of key-switching. As dnum increases from 1 to $L+1$, the overall memory consumption in key-switching and the workload for NTT strictly grows, while the workload for basis conversion decreases. The overall calculation workload also depends on the selection of other parameters, especially multiplicative depth L. Therefore, different dnums are suitable for various scenarios.

The variation of dnum poses generality challenges to the accelerators, especially for those with specific-designed circuits due to the high complexity of changing them. Since the proportion of different operations in the total workload is not static, the first challenge for the designers is to decide how to distribute the limited resources in the accelerator to different calculation units (*i.e.*, NTT and basis conversion) to achieve balanced throughput for all the typical applications.

The second generality problem is how to decide the parallelism scheme. Since NTT and fast basis conversion require different schemes to be efficient, for the accelerators, how to parallel process polynomials is impacted by the relative workload of NTT and basis conversion, which is relevant to dnum. The parallelism schemes adopted by existing works can be categorized into rPLP (residue-polynomial-level parallelism) and CLP (coefficient-level parallelism), two different polynomial access patterns for the accelerators to achieve high parallelism. Under RNS representation (§2.3), we need to perform concurrent operations on multiple residue polynomials. With rPLP, each processor in the accelerator individually executes operations over a single residue polynomial. The parallelism is achieved by distributing multiple residue polynomials to different processors. With CLP, multiple processors in the accelerator collaboratively execute operations for a single residue polynomial, and n coefficients in the same polynomial are distributed to different

processors. There is no superior or inferior of these two schemes in terms of coefficient-wise operations like polynomial addition and dyadic multiplication. However, CLP introduces global data communication across different processors for NTT. Similarly, rPLP leads to extra data exchange in basis conversion. Therefore, both schemes may cause performance degradation in certain operations.

In key-switching, both basis conversion and NTT are dominant. The designer may compare the relative workload of basis conversion and NTT in the targeted applications to choose between CLP and rPLP. For example, when a relatively smaller dnum is chosen, which increases the importance of basis conversion, CLP is a preferred choice. Nevertheless, generality remains to be a challenge.

4.3 Challenges of Accelerating Bootstrapping

In this section, we mainly analyze bootstrapping in word-wise FHE since it is more complicated than that in bit-wise FHE (details in §3.3), and is the focus of the latest FHE accelerators [63, 65, 82]. Bootstrapping in word-wise FHE consists of four complicated steps, making it the most time-consuming operation in word-wise FHE. The impressive overhead comes from the large consumption of multiplicative depth and enormous homomorphic permutations, which both lead to extremely high computation complexity and memory access.

Large Consumption of Multiplicative Depth: As reported in [82], bootstrapping in the LSTM benchmark consumes 61% of the maximum multiplicative depth (35 out of 57 levels). The significant consumption of multiplicative depth mainly comes from the approximation schemes applied in bootstrapping, which include deep calculations.

Therefore, for FHE applications requiring unbounded multiplication depth, such as deep learning training, the maximum multiplicative depth L must be large enough to support bootstrapping. Furthermore, since bootstrapping involves dramatic overhead, we should further increase depth L to use more multiplicative depth to perform application computations before bootstrapping, thus reducing the frequency of bootstrapping during the whole calculation to guarantee the effectiveness of an application.

Consequently, the growth of depth L leads to a larger modulus Q, and the degree of polynomial n should be raised accordingly to reach a certain security level of RLWE [24]. Large modulus and degree of polynomial lead to the growing size of ciphertexts and keys, thus significantly increasing the workload of polynomial calculations and memory consumption.

Enormous Homomorphic Permutations: The second and fourth steps of bootstrapping, CoeffToSlot and SlotToCoeff, suffer from the high computational overhead of homomorphic operations, especially when the plaintext messages are densely packed in the ciphertext. In the implementation of [34], CoeffToSlot on a ciphertext, which packs 4096 plaintext messages, requires $\sim$ 30 homomorphic permutations, leading to large computation complexity. Moreover, homomorphic permutations introduce massive additional switching keys, dramatically inflating the memory space needed and increasing the bandwidth requirements. In detail, the size of a single switching key is about dnum$\times$ the size of a ciphertext, and permutations aiming at different rotation steps need distinct keys, which introduce ignorable memory storage and access overhead. For example, according to the analysis in [63], with $n = 2^{15}$, 40 different switching keys should be pre-computed and stored in preparation for permutations in the linear transformation of bootstrapping.

5 FHE Accelerators

In this section, we will first comprehensively review the existing FHE accelerators in chronological order (§5.1). Then we briefly introduce how existing FHE software support these accelerators (§5.2). Finally, we will summarize their

Name	Hardware Target	Software/Hardware Co-design	Polynomial Parallelism	Supported Schemes			Supported Features		Open-source
				BFV	CKKS	TFHE	Programmable	Bootstrappable	
cuHE [1]	GPU	○	rPLP	○	○	●	○	●	●
cuFHE [4]	GPU	○	rPLP	○	○	●	○	●	●
nuFHE [5]	GPU	○	rPLP	○	○	●	○	●	●
HEAT [80]	FPGA	●	rPLP	●	○	○	○	○	●
HEAX [78]	FPGA	○	rPLP	○	●	○	○	○	○
HEXL [28]	CPU	○	rPLP	●	●	○	●	○	●
HEXL-FPGA [11]	FPGA	○	rPLP	●	●	○	●	○	●
100× [62]	GPU	○	rPLP + CLP	●	●	○	●	●	●
F1 [81]	ASIC	●	rPLP	●	●	○	●	◐	○
CraterLake [82]	ASIC	●	CLP	●	●	○	●	●	○
BTS [65]	ASIC	●	CLP	●	●	○	●	●	○
ARK [63]	ASIC	●	rPLP + CLP	●	●	○	●	●	○
Poseidon [88]	FPGA	●	CLP	●	●	○	●	●	○
FAB [23]	FPGA	●	CLP	●	●	○	○	●	○
TensorFHE [51]	GPU	●	rPLP	●	●	○	●	●	○

Table 2. Overview of all FHE accelerators (in chronological order).

evolution trajectory to address particular challenges mentioned in the previous section, which establishes a qualitative connection among them (§5.3).

Table 2 provides an overview of FHE accelerators by demonstrating the hardware leveraged, the data parallelism schemes used, and the features of these FHE accelerators. We also make the following explanations to make it clear. First, the software/hardware co-design denotes whether the design leverages software's flexibility and hardware's efficiency to properly distribute and schedule the workload in the accelerator. In particular, we regard co-design is absent in all CPU-based and GPU-based works, which only include software design based on off-the-shelf hardware architectures. Second, similar to previous works [86], the FHE schemes are grouped into families of related schemes. For example, BFV stands for BFV [50]/BGV [32] and TFHE stands for TFHE [39]/GSW [55]. Third, among the different features of the accelerators, we mainly focus on whether they are programmable and bootstrappable. Programmable refers to the accelerator's ability to support a variety of cryptographic parameters without hardware architecture reconfiguration. Bootstrappable means it is possible to execute bootstrapping and achieve unlimited FHE operations with the accelerator. The half circle (◐) here represents that the accelerator supports bootstrapping, but the performance is far from meeting practical requirements.

Worth noting, although the FHE accelerators rely on the acceleration of polynomial computations, such as NTT, it is not reasonable to categorize works that only accelerate these basic modules as FHE accelerators. For example, FFT accelerators are commonly applied in other fields, such as signal processing, but most of them cannot be directly used to boost the performance of FHE schemes. Our survey will not cover them but focus on accelerators that explicitly improve the efficiency of FHE schemes.

5.1 Survey of Existing FHE Accelerators

For all the FHE accelerators, the common building block is the FFT/NTT module, as it is the bottleneck of polynomial calculations and directly determines the performance. We need to point out that most existing accelerators (except for the GPU-based design nuFHE [5]) do not implement both FFT and NTT computations since FFT contains floating-point operations, while NTT only includes integer operations. They require completely different hardware circuits for acceleration. Therefore, to simplify the design complexity and reduce the hardware resource overhead, the accelerator

designers usually accelerate one of FFT and NTT, and therefore can only support FHE algorithms using the corresponding operations, as we listed in Table 2.

By connecting the FFT/NTT module with other computation units with optimized data paths, the accelerators can finish their target FHE jobs. Different works have variant designs for FFT/NTT modules and data paths, targeting their acceleration goals. For earlier works, since their computation scenarios are limited, they usually have fixed designs for NTT/FFT computation units and fixed calculation workflow to reduce design complexity. As the application scenarios continue to increase, later works tend to design programmable NTT/FFT units and data paths to support different parameter settings and various applications. Moreover, later works find it is important to introduce more on-chip memory spaces with customized devices, as the memory of general hardware like GPU and FPGA becomes insufficient because the FHE applications become more complicated and require more calculations. As a result, Application-Specific Integrated Circuit (ASIC) becomes a preferred choice for designers who can afford it.

5.1.1 cuHE/cuFHE CUDA Homomorphic Encryption Library (cuHE) was proposed by Dai *et al.* in 2015 [1, 45]. cuHE uses GPU for acceleration and provides CUDA implementation of NTT and CRT. CUDA-accelerated Fully Homomorphic Encryption Library (cuFHE) was proposed by Dai *et al.* in 2018 [4]. cuFHE leverages implementations of cuHE to boost the performance of TFHE [39]. Both cuHE and cuFHE are open-sourced standalone acceleration libraries [1, 4] and do not provide official integration with FHE libraries. However, they only support accelerating polynomial operations with NTT, while an FFT-based implementation may achieve higher performance for TFHE. Moreover, besides limited functions, cuFHE adopts fixed cryptographic parameters, which cannot be configured to achieve high generality.

5.1.2 nuFHE GPU-powered Torus FHE implementation (nuFHE) was launched by NuCypher in 2018 [5]. Similar to cuHE/cuFHE, nuFHE also adopts GPU to accelerate FHE schemes. However, different from them, nuFHE provides either FFT or NTT to improve performance. nuFHE is an open-source standalone library [5] and provides Python APIs. With FFT, nuFHE achieves better performance compared with cuFHE. But it shares the similar disadvantage of limited generality.

5.1.3 HEAT HEAT was proposed by Roy *et al.* in 2019 [80]. Different from cuHE, cuFHE, nuFHE, HEAT targets accelerating a word-wise FHE scheme: BFV. Moreover, besides general hardware which has been used in previous works, HEAT further leverages FPGA to achieve a more flexible hardware design. Thus, HEAT utilized a heterogeneous ARM-FPGA co-designed architecture implemented on Xilinx ZCU102 Evaluation Kit [7]. The authors further migrated HEAT to the f1 instance of Amazon AWS in 2020 [85]. Both implementations are open-sourced. The implementation of HEAT on ZCU102 comprises several hardware coprocessors on FPGA and a multi-core (4 cores) ARM processor. The parallel coprocessors can efficiently execute primitive operations in BFV, including addition, subtraction, multiplication, modulus switching, and NTT. The ARM cores control the coprocessors' workflow and manage the network connection to the applications. By using the software as a workflow controller, HEAT can perform polynomial arithmetics and homomorphic operations like key-switching in BFV by combining different primitive operation units.

Considering the limited on-chip memory (BRAM/URAM) and calculation units (DSP), it is impossible to implement a fully pipelined NTT processor on FPGA. Instead, the authors instantiated two CT butterfly units for each NTT core and accomplished the NTT by reusing them. This design is a compromise solution due to limited resources, thus inevitably leading to non-optimal performance. Another problem caused by resource constraints is the relatively smaller polynomial degree. The polynomial degree supported by HEAT is 4096, which does not fulfill most practical requirements and significantly limits its generality.

5.1.4 HEAX HEAX was proposed by Microsoft in 2020 [78]. It provides a highly performant hardware architecture to accelerate the operations of CKKS. HEAX is not an open-source project. The acceleration foundations of HEAX are the primitive modules: NTT and multiplication modules. Each module consists of multiple calculation cores, which could be adjusted to match the required throughput.

Different from HEAT, which depends on the software to manipulate hardware primitive operations (*e.g.*, NTT and multiplication) when implementing key-switching, HEAX implements a specific key-switching module to minimize the overhead of software and hardware interaction. Following the workflow of key-switching, the key-switching module instantiates multiple primitive modules and BRAMs to construct the pipeline. The modules can be adjusted to balance the throughput in the pipeline based on specific cryptographic parameters.

However, the adjustment of HEAX is accomplished by physically modifying the modules. The FPGA has to be reconfigured to adapt to different cryptographic parameters, which limits the generality of the design.

5.1.5 HEXL Unlike previous works relying on specific hardware devices, such as GPU and FPGA, Intel proposed Intel Homomorphic Encryption Acceleration Library (HEXL) in 2021 [28]. HEXL exploits the SIMD features provided by Intel CPUs, which are easily accessible, to provide plug-and-play acceleration capacities for FHE. HEXL has been integrated with PALISADE [19], Microsoft SEAL [17] and HElib [14] by replacing the underlying arithmetic implementations. It is open-sourced, and the code is hosted on Github [16].

With the single-threaded implementation of primitive operations, including NTT/iNTT and dyadic multiplication, based on the Intel AVX-512/AVX512IFMA instructions [10], HEXL reaches high single-core acceleration. Since HEXL is thread-safe, users can achieve better performance by paralleling different operations with multi-threading. However, due to architecture deficiency, the overall performance largely degrades when there are many threads. For example, too many threads with SIMD cause significant heat dissipation. As a result, core frequency is dramatically reduced when reaching the TDP [57]. Moreover, similar to the CPU-based acceleration libraries in other domains, HEXL also suffers from slow memory access, further reducing its overall performance.

5.1.6 HEXL-FPGA To compensate for the disadvantages of HEXL, Intel further proposed HEXL-FPGA in 2021 [11]. HEXL-FPGA offers the HLS-based (high level synthesis) implementation of NTT/iNTT, multiplication, and key-switching. Since HEXL-FPGA is open-sourced, users can compile each of the functions into an individual bitstream and program the FPGA device. HEXL-FPGA can be integrated with aforementioned HEXL to accelerate corresponding FHE libraries. HEXL-FPGA is open-sourced and under active development [11].

However, there are inherent problems with HLS-based solutions. HLS designs are written in high-level language and rely on the compiler to convert the codes into hardware design. Due to the essential difference between the hardware and software, the compiling process may lead to redundant resource consumption and complex workflow synchronization [26], thus leading to suboptimal performance. Therefore, it's hard for HEXL-FPGA to fully utilize the advantages of FPGA. Moreover, HEXL-FPGA only supports a limited range of cryptographic parameters, which will be demonstrated in §6.2.

5.1.7 100× 100× was proposed by Jung *et al.* in 2021[62]. It provides higher acceleration of CKKS than previous works (*i.e.*, HEXL-FPGA and HEAX) with the powerful V100 GPU [3]. The reason is that V100 has more hardware resources due to better semiconductor manufacturing processes and works at a much higher frequency than the FPGAs adopted in previous works. 100× introduces memory-centric optimizations to increase end-to-end performance and achieves over 100× acceleration ratio compared to the single-thread CPU implementation.

The implementation of NTT in 100× is built based on the hierarchical approach in [64]. Its basic idea follows the 4-step FFT/NTT algorithm [27]. The implementation of NTT is divided into two separate kernels, making it possible to cache all input data in the shared memory. Moreover, due to the limited size of registers in GPU, both kernels further decompose the NTT based on a generalized version of the 4-step FFT/NTT algorithm. In addition to the decomposition, schemes including coalesced memory access and on-the-fly twiddle factors generation are leveraged to decrease the overhead in memory access.

However, although the memory accesses in the basic operations have been optimized, the end-to-end performance of a FHE task is still bottlenecked by the bandwidth of the main memory. To relieve the bottleneck, the authors of 100× tend to fuse multiple kernels into a single kernel, which allows the data cached on the chip to be reused by a series of consecutive operations and reduces global memory accesses. Nevertheless, since GPU's architecture is designed for calculations between small numbers (*e.g.*, FP16), it does not offer large on-chip memory. As a result, the intensive memory access still degrades the performance of 100×.

5.1.8 F1 To solve the problems of insufficient resources (*e.g.*, the FPGA-based ones) and unsuitable fixed architecture (*e.g.*, the CPU or GPU-based ones), recent works have shifted to explore the potential of application-specific integrated circuits (ASICs). Following this trend, F1 was proposed by Feldmann *et al.* in 2021 [81]. To pursue more practical acceleration for FHE schemes, F1 is the first programmable FHE accelerator with a dedicated architecture, which denotes it can support several FHE schemes with a large range of cryptographic parameters. F1 is not open-sourced.

At its core, F1 implements 16 computation clusters, each containing several primitive function units (FU), including NTT, modular multiplication, modular addition, and automorphism. Different units can compose high-level FHE operations such as key-switching. All the FUs are pipelined and vectorized to process 128 elements in each cycle. Therefore, polynomials with degrees that are multiples of 128 can be handled by sequentially feeding the operands to the pipeline. Specifically, to implement NTT with the 128-element processor, F1 leverages the 4-step FFT/NTT algorithm [27] to decompose a NTT into multiple vectorized operations with much fewer input elements. F1 tries to minimize data movement overhead by proposing a hierarchical storage system. The off-chip high-bandwidth memory (HBM) is the global memory that directly interacts with the CPU server. A 64MB scratchpad built on 16 banks of SRAM is designed as the on-chip cache and fetches data from HBM. The computation clusters communicate with the scratchpad and store the data used for the current operation in the limited vector registers. The communication between the scratchpad and clusters depends on a complex fully connected network (three 16×16 crossbars).

The design of F1 has three main problems. First, the performance of F1 highly relies on sufficient parallelism among 16 clusters and efficient data movement in the accelerator, which places a high demand on the software compiler for operation scheduling. Considering that FHE computations vary significantly in terms of different workflows and different parameter settings, the compiler is complicated. However, F1 does not provide many details about its compiler. The second problem is that F1 uses a fixed key-switching algorithm (generalized key-switching with dnum = $L + 1$). If the multiplicative depth L of ciphertexts is high, key-switching is extremely slow. Last, F1 only supports non-packed bootstrapping [36], which only works for ciphertext with a single number packed in the polynomial and is far from practical in real-world applications. The reason is that the maximum degree of polynomial that F1 supports via the 4-step FFT/NTT algorithm is 16384, which is too small for F1 to execute fully packed bootstrapping [30] under 80-bit or 128-bit security level of RLWE. Without efficient bootstrapping, F1 struggles to evaluate deep arithmetic functions.

5.1.9 CraterLake CraterLake was proposed by Samardzic *et al.* in 2022 [82]. It is not an open-source project. CraterLake is a follow-up to F1 and targets unbounded-depth homomorphic multiplications. Consequently, CraterLake uses the fully

packed bootstrapping [30] to refresh the multiplicative depth of ciphertexts. To balance the frequency of bootstrapping and the size of ciphertexts, the authors of CraterLake chose the number of multipliers required per homomorphic multiplication as the criterion to evaluate the overall computational complexity of an FHE program. Moreover, the authors followed the idea of the vectorized unit in F1 and logically designed the whole accelerator as a single 2048-element vectorized processor to handle the large ciphertext, which requires the maximum degree of polynomial and multiplicative depth to be 65536 and 60 respectively. However, such a design can not be naively implemented on ASIC considering the complex combinational and sequential logic circuits. Thus, in CraterLake, the 2048-element processor is physically composed of eight 256-element groups.

Similar to F1 [81], CraterLake implements several functional units (FU) in each 256-element group. The difference is the two additional FUs in CraterLake, *i.e.*, Change-RNS-base (CRB) and switching key generator (KSHGen), and both units are introduced to further optimize key-switching. The CRB unit, which mainly executes modular multiplication and modular summation, is specifically designed to accelerate fast basis conversion when dnum is relatively small (*e.g.*, dnum = 1). To save additional memory space, the KSHGen unit generates switching keys on the fly, which were pre-computed and cached in the device memory in previous works, such as F1 [81] and HEAX [78].

Another major difference between CraterLake and previous works is that CraterLake adopts CLP as the data parallelism scheme rather than rPLP due to the increasing workload of fast basis conversion. With the CLP scheme, CraterLake only processes one polynomial operation simultaneously, and the coefficients of polynomials are distributed to the 256-element groups with a static distribution strategy, leading to three significant advantages. First, as a particular coefficient can only be assigned to a specific group, the complex 16×16 crossbar in F1, which deals with the dynamic data exchange between on-chip scratchpad and groups, is no longer required. Data movement caused by operations among different polynomials can be eliminated as well. Second, the parallelism of CraterLake is not influenced by the varying multiplicative depth or numbers of concurrent operations, which maintains the performance throughout an entire FHE program and simplifies the software scheduler.

Although CraterLake is programmable enough to support different cryptographic parameters, the FUs, especially NTT units, may face functionality limitations or resource underutilization because of the static pipelined circuit, as we mentioned in §4.1.3. The computation resources in CraterLake can be fully utilized if and only if $n = 65536$, which is a common problem for works containing a pipelined NTT circuit (*e.g.*, F1 and ARK). Besides, in CraterLake, the CRB unit occupies 34% on-chip area, reducing the performance of other operations, like NTT. Therefore, in the scenarios where larger dnum is optimal, the CRB units are underutilized and CraterLake cannot deliver sufficient acceleration.

5.1.10 BTS BTS was proposed by Kim *et al.* in 2022 [65]. It is a follow-up to 100× and shares similar design goals with CraterLake, *i.e.*, a bootstrappable and programmable hardware architecture for word-wise FHE.

Because of the similar goal, many optimization approaches in BTS are close to those in CraterLake, including the basis conversion unit, CLP parallelism scheme and on-the-fly data generation. The most significant difference between CraterLake and BTS is the pattern in which they place basic functional units, thus leading to different implementations of basic operations. The architecture of BTS is close to that of modern GPUs, consisting of a 64×32 two-dimensional (2D) array of processing elements (PE). Each PE comprises a basis conversion unit and a 2-point NTT/iNTT unit (*i.e.*, a single butterfly unit). The 2048 PEs are interconnected via vertical and horizontal crossbars. An example of 131072-point NTT with the architecture works as follows. First, each PE performs a 64-point NTT independently. Second, 64 PEs in the same row perform a 64-point NTT with data synchronization through horizontal crossbars. Third, 32 PEs in the same column perform a 32-point NTT with data synchronization through vertical crossbars. This process generalized

the 4-step FFT/NTT algorithm. Each PE plays a similar role to the GPU threads, and the crossbars accomplish thread synchronization. To facilitate the unique computational pattern, the authors placed multiple smaller local scratchpads in each PE instead of putting a large global one that communicates with all the PEs. The local scratchpad is directly connected with HBM and eliminates additional overhead from the hierarchical data distribution.

However, a potential drawback of BTS is that the 2D array structure cannot execute NTT with a fully pipelined workflow. As mentioned before, NTT operation in BTS consists of three sequential steps and all the PEs are involved in each step. Therefore, no pipeline parallelism can be achieved if multiple NTT operations should be processed, which may decrease the end-to-end performance. Similar to F1 and CraterLake , BTS is not open-sourced.

5.1.11 ARK ARK was proposed by Kim *et al.* in 2022 [63]. It consists of four vectorized processing clusters, similar to the 256-element groups of CraterLake. Each cluster contains several primary operation units. Different from previous works that target accelerating bootstrapping only from hardware angle [65, 82], ARK is the first work that analyzes and modifies the bootstrapping algorithm, thus achieving an algorithm and hardware co-design to optimize the accelerator's performance. Specifically, the authors proposed multi-hop homomorphic rotation and on-the-fly residue extension to significantly reduce the size of switching keys and plaintexts used in bootstrapping, respectively, which lowers the difficulties in hardware design.

As mentioned earlier in this section, the workload of basis conversion is heavy in bootstrapping, making CLP a preferred parallelism scheme for bootstrappable accelerators like CraterLake and BTS. But CLP also introduces communication overhead in NTT. According to ARK, NTT and basis conversion separately take up 54.8% and 34.2% of the overall computational workload given practical bootstrappable parameters. Since both functions are crucial to end-to-end performance, ARK leverages both schemes, *i.e.*, CLP for basis conversion and rPLP for the other operations including NTT. A switching mechanism between the two data distribution patterns is also used by ARK, especially for functions that contain both NTT and basis conversion, such as key-switching. In ARK's paper, this hybrid parallelism is proven more efficient than rPLP, but whether it is better than CLP is not discussed.

5.1.12 Poseidon Poseidon was proposed by Yang *et al.* in 2023 to design a FPGA-based FHE accelerator that could support bootstrapping [88]. Compared to previous FPGA-based accelerators, Poseidon leverages several optimization operations, such as radix-based NTT, optimized automorphism, *etc.*, to improve the resource efficiency of its implementation. Furthermore, Poseidon utilizes a more modern FPGA, U280 [21], with high-bandwidth memory (HBM) [9] support. Therefore, Poseidon manages to implement the complicated bootstrapping algorithm on FPGA. However, Poseidon should still suffer the limitations of aforementioned FPGA-based solutions.

5.1.13 FAB FAB was proposed by Agrawal *et al.* in 2023 [23]. Similar to Poseidon [88], FAB is a FPGA-based FHE accelerator that can support bootstrapping [23]. FAB optimizes the on-chip memory access based on the workflow of bootstrapping algorithm, to eliminate the bottleneck caused by memory access. Similar to Poseidon [88], FAB also uses U280 [21] FPGA with HBM support.

5.1.14 TensorFHE TensorFHE was proposed by Fan *et al.* in 2023. By transforming NTT into multiple sequential general matrix multiplications, TensorFHE can use the Tensor Core Units (TCU) [56] in modern GPUs to accelerate the NTT computation [51]. The core idea of TensorFHE is to leverage fine-grained operation batching to utilize the data parallelism features provided by a GPU. With state-of-the-art GPUs, such as V100 [3] and A100 [12], the authors claim that TensorFHE can outperform other accelerators implemented with CPU, GPU and FPGA. TensorFHE could achieve comparable performance as F1 [81] in some scenarios, due to the latest semi-conduct technologies adopted by

these modern GPUs. TensorFHE should still suffer from the aforementioned non-optimal acceleration due to GPU's architectural deficiency.

5.2 Upper-layer Software Support

To effectively utilize the accelerators mentioned above, proper upper-layer software support is crucial. However, integrating FHE software with these accelerators poses significant challenges. One key obstacle is the need to establish a well-defined and unified interface that accommodates different FHE accelerators, each utilizing distinct data structures for input/output and requiring diverse workflows. Addressing this challenge is signifant to the research community, as achieving transparency of FHE accelerators to upper-layer software is essential for practical real-world deployment.

A promising advancement towards solving this problem is the OpenFHE project [25]. OpenFHE is an open-source project that offers efficient and extensible implementations of the latest FHE schemes. Notably, OpenFHE introduces a hardware abstraction layer (HAL) designed to support CPU, GPU, FPGA, and ASIC-based FHE accelerators. The HAL provides C++ abstract classes that enable accelerators to implement their own backend logic. These classes encompass essential mathematical operations (such as NTT, iNTT, vector addition, *etc.*) and lattice/polynomial layer functionalities (including RNS subroutines). It is worth mentioning that OpenFHE already includes a hardware backend implementation for HEXL [18, 28].

By addressing the interface and compatibility challenges and offering a flexible HAL, OpenFHE empowers researchers and developers in the FHE community to seamlessly integrate diverse FHE accelerators into their upper-layer software solutions. We believe that this advancement will shed light on bridging the gap between FHE applications and accelerators.

5.3 Observations on Evolution of Existing Works

Based on the above survey, we make the following observations on the evolution of existing FHE accelerators and the underlying connection among them. We also illustrate how they address particular challenges mentioned in §4.

From General Hardware to Application-Specific Integrated Circuit: Most latest FHE accelerators choose Application-Specific Integrated Circuit (ASIC) as their hardware platform for two reasons. First, general-purpose hardware, such as CPU and GPU, suffer from fixed architecture and limited on-chip memory, which fail to address the challenges mentioned in §4. Second, FPGA suffers from relatively limited programmable resources and low operational frequency, which restricts it from reaching high performance. Although some latest FHE accelerators still utilize FPGA or GPU in their design [23, 51, 88], they still suffer the mentioned limitations, thus leading to suboptimal performance.

The advantage of applying ASICs to FHE accelerators is that designers can optimize the hardware architecture with high flexibility and utilize state-of-the-art technologies of the computer architecture community according to the requirement of FHE computations. Recent ASIC-based works adopt specially designed architecture, such as crossbars, to enable complex polynomial operations described in §4.1.1. They also use high-speed memory, including HBM and SRAM, to accelerate the intensive memory access mentioned in §4.1.2 and §4.3. While promising, this design choice also introduces several problems: 1) most modern ASIC-based FHE accelerators are expensive to produce, especially when the chip size is up to hundreds of square millimeters, *e.g.*, a 14nm ASIC of 100mm^2 requires millions of US dollars to tape out; 2) most of these ASIC-based solutions are not open-sourced due to reasons such as IP restrictions, making them difficult to reproduce, thus undesirable for the research community.

Software/Hardware Co-design: Early FHE accelerators perform certain simple operations, such as NTT, after receiving a single instruction from upper-layer software without any instruction scheduling [78, 80]. Since these operations follow the static hardware workflow, which cannot be dynamically adapted, the accelerators suffer from limited generality (§4.1.3). Moreover, when accelerating an end-to-end application containing massive high-level operations such as bootstrapping (§4.3), the overhead of frequent interactions between software and hardware cannot be ignored.

To overcome these problems, recent FHE accelerators adopt software/hardware co-designed approaches. Specifically, they first leverage hardware-friendly algorithms to divide the hardware resources into multiple computation clusters which can be independently scheduled [63, 65, 81, 82]. Furthermore, the designers can make extensive use of the software's flexibility to apply adaptations according to the specific application to efficiently support complicated end-to-end applications.

However, the current software design of FHE accelerators fails to leverage knowledge from some of the latest techniques, *e.g.*, various FHE compilers [38, 46]. We identify it as a potential future direction in §7.3.

Enhanced Programmability: Programmability, *i.e.*, supporting different cryptographic parameters without hardware architecture reconfiguration, is not achieved in most of the early works due to the following two reasons. First, early works rarely focus on end-to-end acceleration for real-world applications. Thus, they do not have such a need. Second, programmability is not easy to achieve, considering the complexity of FFT/NTT.

Recent FHE accelerators have begun to adopt the 4-step FFT/NTT algorithm not only to increase parallelism but also to improve the architecture's programmability to handle different parameters, specifically various n. Readers may refer to Appendix B.2 for more details. The improved programmability alleviates the generality limitation in polynomial computations described in §4.1.3, although the problems are not eradicated. Currently, an unresolved issue is the generality challenge in key-switching (§4.2). Recent works focusing on deep calculations show unoptimized performance in shallow tasks. As described in [82], in shallow benchmarks without bootstrapping (L is between 4 and 8), CaterLake is slower than F1 due to the underutilization of the basis conversion units that occupy a large proportion of hardware resources.

Unlimited Depth of Operations: Interestingly, the latest few works (*i.e.*, CraterLake [82], BTS [65], and ARK [63]) show a similar tendency of accelerating bootstrapping to achieve unlimited 'fully' homomorphic operations. This common goal leads to the convergence of multiple design choices. First, optimized algorithms, including 4-step FFT/NTT and generalized key-switching, are preferred because of their advantages in handling large ciphertexts. Recent studies are willing to allocate a considerable amount of hardware resources to corresponding structures such as global transpose and fast basis conversion. Second, recent works invest much effort in eliminating the memory access bottleneck. They tend to reduce the memory overhead at the cost of introducing extra calculations, like on-the-fly generation of essential parameters. Last but not least, since all these works have emphasized the importance of basis conversion, CLP is becoming the primary parallelism scheme, contrasting with rPLP in the earlier designs.

6 Evaluation

6.1 Evaluation Methodology

In this section, we provide a quantitative comparison of these existing FHE accelerators. As introduced in §5, some FHE accelerators are open-sourced. We will use our testbed to reproduce the results of some representative ones. Since these open-sourced FHE accelerators do not provide direct support for end-to-end algorithms, we mainly evaluate how they

accelerate the performance of NTT and key-switching. In this paper, we evaluate HEXL, HEXL-FPGA, and 100× as word-wise FHE schemes. We also evaluate two bit-wise FHE accelerators: cuFHE and nuFHE.

Second, since some latest FHE accelerators are not open-sourced, we will directly use the results from their original papers. For example, the paper of HEAX and F1 provides its performance of accelerating NTT and key-switching while TensorFHE also provides its performance of NTT, thus we will align these results with our testbed results. Moreover, some latest accelerators, such as F1, BTS, ARK, Poseidon, FAB and TensorFHE provide end-to-end performance results for accelerating real-world applications, we will also include these results in our paper. Specifically, we demonstrate the performance of two deep applications, *i.e.*, ResNet20 [67] and Logistic Regression [59], and two shallow applications, *i.e.*, LoLa-MNIST and LoLa-CIFAR [33]. Deep applications contain many ciphertext multiplications and therefore require bootstrapping, while shallow applications only include limited multiplications and do not need bootstrapping. For the detailed settings of the end-to-end tests, we refer the readers to the original papers of the accelerators.

Testbed settings: We use a single X86 server as our testbed. The server is equipped with an Intel(R) Xeon(R) Gold 5115 CPU running at 2.40GHz and 128GB RAM. The CPU supports AVX-512 FMA [10]. The operating system is Ubuntu 18.04.5 LTS. We also use Intel PAC D5005 Acceleration Card with an Intel Stratix 10 GX FPGA [15] to reproduce the results of HEXL-FPGA. For all accelerators that require the GPU as their target hardware, we use NVIDIA V100 GPU with 32GB RAM for evaluation [3]. All the experiments are evaluated in a docker environment with the docker version 20.10.21.

6.2 Evaluation Results

NTT/FFT. First, we will evaluate the performance of word-wise FHE accelerators. Figure 5 shows the NTT performance of HEXL, HEXL-FPGA, 100×, HEAX, F1 and TensorFHE. Please note that the performance of HEXL, HEXL-FPGA and 100× is measured on our testbed, while the performance results of HEAX, F1 and TensorFHE are from their original paper. We also run SEAL [17] without any accelerators to demonstrate the baseline performance (denoted as No Acc in the Figure). In this evaluation, we use three settings of n, *i.e.*, $n = 4096, 8192, 16384$. Similar to HEAX, we set the bit-width of polynomial coefficients in NTT to 52 for evaluation [78]. But F1 chooses the bit-width of 32 as it is the largest word size in F1. Theoretically, the performance of F1 should be slightly worse if the bit-width is 52.

We mainly have the following observations. 1) when the FHE accelerators leverage more advanced hardware technologies, the performance is largely improved. For example, the CPU-based accelerator, HEXL, can only achieve up to 3.0× acceleration ratio, while ASIC-based accelerators, F1, can achieve up to 20546.9× acceleration ratio. 2) contradicting our common wisdom, specific-designed hardware-based solutions do not always yield better performance than general hardware-based ones. For example, HEXL-FPGA and HEAX cannot achieve a better acceleration ratio over 100× and TensorFHE. The core reason is that HEXL-FPGA and HEAX adopt FPGA as their hardware platform, which suffers from the aforementioned problems such as limited programmable resources and low working frequency. Precisely, V100 GPU in our evaluation has the peak performance of $\sim$ 250 INT8 TOPS with tensor core [3], which is $\sim$ 10× better than Stratix 10 FPGA [2] used in HEAX and HEXL-FPGA.

Figure 7 shows the performance of NAND gate achieved by cuFHE and nuFHE on our testbed. The NAND gate is a typical example that includes bootstrapping in bit-wise FHE schemes, such as TFHE. Polynomial multiplication in TFHE can be accelerated with either NTT or FFT, and we mark the schemes used by different libraries in the Figure. For the baseline (No Acc), we run the TFHE software libraries [20].

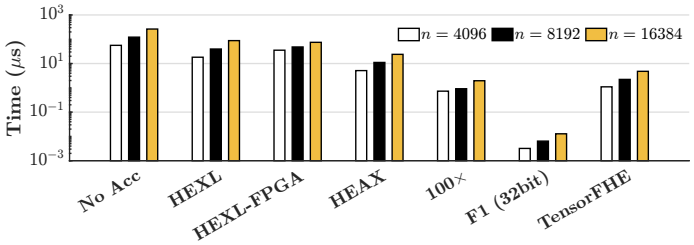

Fig. 5. Performance of NTT. The performance of HEXL, HEXL-FPGA and 100× is measured on our testbed while the performance results of HEAX and F1 are from their original paper.

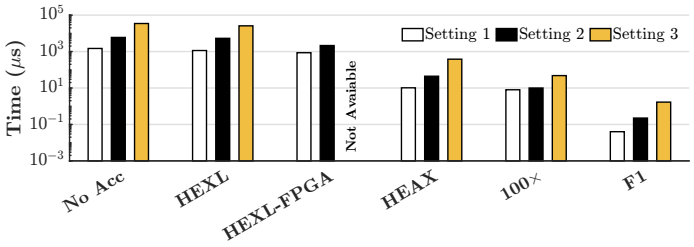

Fig. 6. Performance of key-switching. The performance of HEXL, HEXL-FPGA and 100× is measured on our testbed while the performance results of HEAX and F1 are from their original paper. HEXL-FPGA does not support Setting3.

From our evaluation, we can observe that nuFHE and cuFHE can achieve 170.6× and 186.2× acceleration with NTT implementation, respectively. Moreover, nuFHE also supports using FFT to accelerate the NAND gate, which can achieve up to 432.0× acceleration.

Key-switching. In our evaluation, we use three settings.
Setting1: $n = 4096$, $L = 1$, $\log(P \cdot Q) = 109$, dnum = 2,
Setting2: $n = 8192$, $L = 3$, $\log(P \cdot Q) = 218$, dnum = 4,
Setting3: $n = 16384$, $L = 7$, $\log(P \cdot Q) = 438$, dnum = 8.
Figure 6 shows the results. We have similar results as previous FFT/NTT experiments. Worth noting, HEXL-FPGA cannot perform key-switching with the most complicated setting, *i.e.* setting3, which confirms the potential drawback as discussed in §5.1.6.

End-to-end deep applications. Figure 8 shows the end-to-end performance over two applications achieved by 6 latest accelerators, *i.e.* F1, BTS, ARK, Poseidon, FAB and TensorFHE. Since these solutions are not open-sourced, we use the results from their original papers. We do not include the results of ResNet20 with F1, 100× and FAB since they are not presented in the original papers. The results reveal that: 1) As discussed in §5.1.8, in a task requiring large multiplicative depth, F1 shows critical performance deficiencies despite its significant performance of NTT. For example, it takes F1 1024ms to execute a single iteration of LR, which is even slower than 100× (775ms), Poseidon (73ms), FAB (103ms) and TensorFHE (222ms); 2) For deep benchmarks, ASIC designs that support efficient bootstrapping achieves a significant increase in performance compared to FPGA and GPU designs. For example, BTS is 2.6× and 7.8× faster than FAB and TensorFHE with LR, respectively. As ARK proposes further improvement over bootstrapping, including algorithm optimization and architecture co-design, it achieves better performance when bootstrapping dominates the workflow. In ResNet20, where bootstrapping takes up to 76.2% of the total workload, ARK achieves a 6.5× speedup compared to BTS.

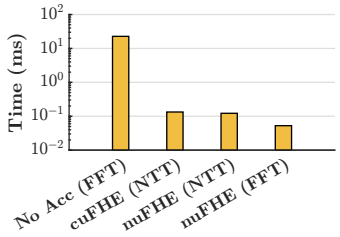

Fig. 7. Performance of NAND gate with NTT/FFT.

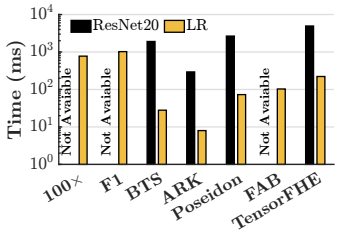

Fig. 8. Performance of end-to-end applications.

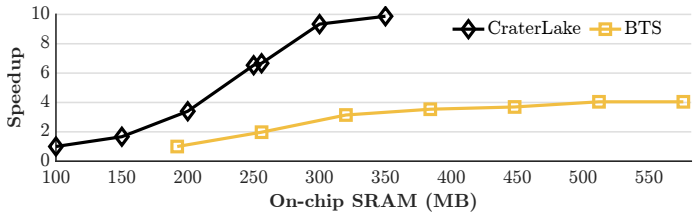

Fig. 9. Performance of Bootstrapping over increasing on-chip SRAM volume.

End-to-end shallow applications. Recent works like BTS, ARK, Poseidon, *etc.*, focus on the acceleration of deep workloads and do not cover much discussion of shallow applications. Since shallow applications are also widely used in real world, here, we briefly discuss the performance of F1 in the shallow workloads. In shallow applications, F1 achieves sufficient acceleration as bootstrapping is not required. In LoLa-CIFAR with unencrypted model weights, F1 achieves $5,011\times$ speedup compared to the CPU. In LoLa-MNIST, F1 is $17,412\times$ and $15,086\times$ faster than CPU when the model weights is unencrypted and encrypted, respectively. Subsequent works like CraterLake don't have a significant advantage over F1 in shallow tasks as they allocate a significant amount of hardware resources to operations specific to deep applications, such as bootstrapping.

Impact of On-chip storage. The performance of ASIC-based FHE accelerators can be significantly influenced by the size of on-chip memory storage [63, 65, 81, 82]. Optimal acceleration is achieved by storing data in on-chip memory to minimize frequent data input/output between on-chip and off-chip memory. However, FHE schemes, particularly during key-switching operations, require a large amount of data, such as ciphertexts and switching keys. If the on-chip memory cannot accommodate all of this data, extensive data communication and exchange occur, resulting in a significant portion of the execution time.

Figure 9 illustrates the impact of increasing on-chip SRAM memory on bootstrapping performance using CraterLake [82] and BTS [65]. These accelerators exhibit different internal designs and implementations, leading to varying behaviors as the volume of SRAM increases. However, a general trend can be observed: performance initially improves and then stabilizes with additional on-chip SRAM. This is because larger on-chip SRAM can store more intermediate data, reducing the frequency of costly data communication, even with high-speed HBM. Nevertheless, once the on-chip SRAM reaches a sufficient size where no further data exchange is required, additional SRAM does not contribute to further performance gains.

7 Discussion on Future Directions

Inspired by the above qualitative (§5) and quantitative (§6) studies, in this section we discuss some future directions of designing and implementing FHE accelerators.

7.1 Application-driven Design Approach

Most existing FHE accelerators are trying to build increasingly powerful accelerators by leveraging more advanced architecture technologies, such as better semiconductor manufacturing processes [63, 65, 78, 81, 82]. While such design methods can largely improve the performance of FHE schemes, they also become more unaffordable, especially for these ASIC-based solutions.

We envision that a future opportunity for FHE accelerators could be the application-driven design approach. The reason is that for some practical FHE applications, the performance is determined not only by the accelerators with extremely high FHE computing performance, but also by the whole architecture stack that customizes with the application. Table 3 provides a brief overview of these works. For example, FHE is widely adopted in private set intersection (PSI), and a recent work called INSPIRE co-designs the storage architecture with the FHE accelerator to improve the end-to-end performance of the PSI application by minimizing the overhead of data movement between the storage controller and FHE accelerators [68]. Cheetah [77] and FxHENN [89] are two works targeting accelerating HE-based CNN inference applications by co-designing CNN inference workflows and HE algorithms. We believe such an application-driven design approach could better balance the application requirements, costs, and design difficulties, providing a promising direction for the future.

7.2 Supporting both Word-wise & Bit-wise FHE Schemes

As discussed in §3, word-wise FHE is more suitable for polynomial evaluation, while bit-wise FHE is preferred for non-polynomial evaluation. However, real-world applications, such as machine learning tasks, require both polynomial and non-polynomial evaluations to be effective. Moreover, ciphertexts switch between the two forms (*e.g.*, CKKS and FHEW in [69]) is extremely complicated, thus time-consuming. Therefore, designing a FHE accelerator that can simultaneously support both word-wise and bit-wise FHE schemes, and further perform efficient ciphertext switch and evaluation is important for real-world applications, which is worthy of future investigation.

7.3 Enhanced Software/Hardware Co-design

In recent works, the software has been considered an important part of the design. Some works, such as F1 [81], CraterLake [82], use compilers for a software/hardware co-design solution. However, they do not provide a comprehensive description of the software design. We believe fully functional compilers should be co-designed with general FHE compilers, such as EVA [46]. As the FHE compiler SoK paper [86] indicates, general FHE compilers can optimize FHE programs based on the cost model of FHE schemes. We believe these general FHE compilers could be integrated with FHE accelerator compilers for improved end-to-end performance by considering a cost model from a hardware-level perspective, which points to a potential future direction.

7.4 From Scale-up to Scale-out

Current FHE acceleration solutions mostly focus on scale-up, *i.e.*, improving the performance of a single accelerator vertically. However, scale-out, *i.e.*, connecting multiple FHE accelerators via networking horizontally, should also be an

Name	Employed Hardware	Supported Application	HE Schemes
INSPIRE [68]	ASIC	PIR	BFV
Cheetah [77]	ASIC	CNN Inference	BFV
FxHENN [89]	FPGA	CNN Inference	CKKS

Table 3. Application-specific FHE Accelerators.

Employed Hardware	AES (s)	Prince (s)
CPU [48]	55 (baseline)	3.3 (baseline)
GPU [44]	7.3 (7.5× ↑)	1.28 (2.58× ↑)
ASIC [75]	0.44 (125× ↑)	0.05 (66× ↑)

Table 4. Accelerators for NTRU-based FHE schemes.

effective way to further improve the performance of FHE applications. As discussed in §4, the data inflation problem has posed a challenge to efficient data movement between on-chip and off-chip memories. This challenge also exists when we connect multiple FHE accelerators via networking in a scale-out manner. Thus, FHE accelerator and network co-design, *e.g.*, integrating both FHE acceleration functions and high-performant networking controllers on the same chip, should be a potential research direction in the future.

7.5 Accelerating NTRU-based Schemes

In this paper, we do not cover accelerators for traditional NTRU-based solutions in details since (1) there are only few literatures targeting at designing accelerators for NTRU-based FHE schemes and (2) these NTRU-based schemes were believed to be vulnerable to attacks and thus impractical [61]. However, recently various modern NTRU-based solutions have been proposed to overcome its original security problems [29]. Moreover, NTRU-based solutions have the advantages of low memory consumption and fast computation. Therefore, we believe that designing practical accelerators for modern NTRU-based solutions also deserves future exploration.

We provide a brief summary of the performance of existing accelerators for NTRU-based schemes using GPUs [44] and ASICs [75]. Following the conventions of these works, we use the homomorphic evaluation of AES and Prince as the performance metric [48]. The CPU implementation in [48] serves as the baseline for comparison. Since these NTRU-based accelerators are not open-sourced, we directly report the data from [44, 48, 75], and the results are presented in Table 4. Compared to the CPU implementation, the GPU and ASIC-based accelerators show performance improvements ranging from 2.6× to 7.3× and 66× to 125×, respectively.

One observation is that the acceleration ratio of NTRU-based accelerators, when compared to accelerators designed for CKKS/BFV/TFHE in §5.1, is relatively lower. For instance, CraterLake can outperform the CPU by five orders of magnitude, while the latest accelerator for NTRU-based schemes [75] achieves only three orders of magnitude speedup. We identify two potential reasons for this discrepancy: (1) NTRU-based schemes themselves are reported to be faster than RLWE-based FHE schemes, leaving less room for further acceleration; (2) The latest ASIC-based accelerator for NTRU schemes is currently simpler than the accelerators discussed in §5.1, indicating potential for further improvements.

8 Conclusion

This paper presents a comprehensive systematization of knowledge through qualitative and quantitative analysis of 14 existing fully homomorphic encryption (FHE) accelerators and their evolution process. We have identified four key trends in the development of these accelerators. First, there is a growing emphasis on leveraging advanced hardware to achieve better acceleration performance. Second, a software/hardware co-design approach is commonly used. Third, there is a focus on enhanced programmability. Fourth, the support for an unlimited number of multiplications has improved through better bootstrapping operation support. Our survey aims to provide researchers with a comprehensive understanding of the current state of FHE accelerators. Additionally, we discuss potential future directions for designing and implementing FHE accelerators, such as developing accelerators for NTRU-based FHE schemes and considering a scale-out methodology instead of scale-up. We hope these discussions will inspire the research community and illuminate the future development of FHE accelerators.

References

[1] 2015. CUDA Homomorphic Encryption Library (cuHE). https://github.com/vernamlab/cuHE. Accessed: 2022-07-30.

[2] 2016. Intel Stratix 10 GX/SX Product Table. https://www.intel.com/content/www/us/en/content-details/652478/intel-stratix-10-gx-fpga-and-intel-stratix-10-sx-soc-fpga-family-overview-product-table.html. Accessed: 2023-03-07.

[3] 2017. NVIDIA V100 Datasheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf. Accessed: 2023-03-07.

[4] 2018. CUDA-accelerated Fully Homomorphic Encryption Library (cuFHE). https://github.com/vernamlab/cuFHE. Accessed: 2022-07-30.

[5] 2018. A GPU implementation of fully homomorphic encryption on torus. https://github.com/nucypher/nufhe. Accessed: 2022-07-07.

[6] 2019. General Data Protection Regulation. https://gdpr-info.eu. Accessed: 2022-10-20.

[7] 2019. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html. Accessed: 2023-03-07.

[8] 2020. HElib Country Lookup Example. https://github.com/homenc/HElib/tree/master/examples/BGV_country_db_lookup. Accessed: 2023-03-07.

[9] 2020. Xilinx Virtex UltraScale+ HBM FPGAs. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html. Accessed: 2023-03-10.

[10] 2021. Intel Advanced Vector Extensions 512 (Intel AVX-512). https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. Accessed: 2023-03-07.

[11] 2021. Intel Homomorphic Encryption (HE) Acceleration Library for FPGAs. https://github.com/intel/hexl-fpga. Accessed: 2022-07-08.

[12] 2021. NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/. Accessed: 2023-03-08.

[13] 2022. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed: 2022-07-07.

[14] 2022. HElib. https://github.com/homenc/HElib. Accessed: 2022-07-31.

[15] 2022. Intel FPGA PAC D5005. https://www.intel.com/content/www/us/en/products/sku/193921/intel-fpga-pac-d5005/specifications.html. Accessed: 2022-10-17.

[16] 2022. Intel HEXL. https://github.com/intel/hexl. Accessed: 2022-11-13.

[17] 2022. Microsoft SEAL. https://github.com/microsoft/SEAL. Accessed: 2022-07-12.

[18] 2022. OpenFHE-HEXL. https://github.com/openfheorg/openfhe-hexl. Accessed: 2023-12-16.

[19] 2022. Palisade homomorphic encryption software library. https://palisade-crypto.org. Accessed: 2022-07-31.

[20] 2022. TFHE: Fast Fully Homomorphic Encryption Library over the Torus. https://github.com/tfhe/tfhe. Accessed: 2022-11-17.

[21] 2022. Xilinx Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html. Accessed: 2023-03-07.

[22] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *ACM Comput. Surv.* 51, 4 (2018), 79:1–79:35. https://doi.org/10.1145/3214303

[23] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. In *The 29th IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 01, 2023*. IEEE.

[24] Martin R. Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin E. Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2019. Homomorphic Encryption Standard. *IACR Cryptol. ePrint Arch.* (2019), 939. https://eprint.iacr.org/2019/939

[25] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[26] Donald G. Bailey. 2015. The advantages and limitations of high level synthesis for FPGA based image processing. In *Proceedings of the 9th International Conference on Distributed Smart Camera, Seville, Spain, September 8-11, 2015*, Ricardo Carmona-Galán and Ángel Rodríguez-Vázquez (Eds.). ACM, 134–139. https://doi.org/10.1145/2789116.2789145

[27] David H. Bailey. 1989. FFTs in external or hierarchical memory. In *Proceedings Supercomputing '89, Reno, NV, USA, November 12-17, 1989*, F. Ron Bailey (Ed.). ACM, 234–242. https://doi.org/10.1145/76263.76288

[28] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*. WAHC@ACM, 57–62. https://doi.org/10.1145/3474366.3486926

[29] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. 2022. FINAL: Faster FHE instantiated with NTRU and LWE. *IACR Cryptol. ePrint Arch.* (2022), 74. https://eprint.iacr.org/2022/074

[30] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12696)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer, 587–617. https://doi.org/10.1007/978-3-030-77870-5_21

[31] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2020. CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. *J. Math. Cryptol.* 14, 1 (2020), 316–338. https://doi.org/10.1515/jmc-2019-0026

[32] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3 (2014), 13:1–13:36. https://doi.org/10.1145/2633600

[33] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low Latency Privacy Preserving Inference. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 812–821. http://proceedings.mlr.press/v97/brutzkus19a.html

[34] Hao Chen and Kyoohyung Han. 2018. Homomorphic Lower Digits Removal and Improved FHE Bootstrapping. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10820)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, 315–337. https://doi.org/10.1007/978-3-319-78381-9_12

[35] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1243–1255. https://doi.org/10.1145/3133956.3134061

[36] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2018), 153. http://eprint.iacr.org/2018/153

[37] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. 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 (Lecture Notes in Computer Science, Vol. 10624)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[38] Eduardo Chielle, Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. E^3: A Framework for Compiling C++ Programs with Encrypted Operands. *IACR Cryptol. ePrint Arch.* (2018), 1013. https://eprint.iacr.org/2018/1013

[39] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[40] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301. https://doi.org/10.1090/S0025-5718-1965-0178586-1

[41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[42] David Bruce Cousins, John Golusky, Kurt Rohloff, and Daniel Sumorok. 2014. An FPGA co-processor implementation of Homomorphic Encryption. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014.* IEEE, 1–6. https://doi.org/10.1109/HPEC.2014.7040950

[43] David Bruce Cousins, Kurt Rohloff, and Daniel Sumorok. 2017. Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor. *IEEE Trans. Emerg. Top. Comput.* 5, 2 (2017), 193–206. https://doi.org/10.1109/TETC.2016.2619669

[44] Wei Dai, Yarkin Doröz, and Berk Sunar. 2014. Accelerating NTRU based homomorphic encryption using GPUs. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014.* IEEE, 1–6. https://doi.org/10.1109/HPEC.2014.7041001

[45] Wei Dai and Berk Sunar. 2015. cuHE: A Homomorphic Encryption Accelerator Library. In *Cryptography and Information Security in the Balkans - Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9540)*, Enes Pasalic and Lars R. Knudsen (Eds.). Springer, 169–186. https://doi.org/10.1007/978-3-319-29172-7_11

[46] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 546–561. https://doi.org/10.1145/3385412.3386023

[47] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. 2015. Accelerating Fully Homomorphic Encryption in Hardware. *IEEE Trans. Computers* 64, 6 (2015), 1509–1521. https://doi.org/10.1109/TC.2014.2345388

[48] Yarkin Doröz, Aria Shahverdi, Thomas Eisenbarth, and Berk Sunar. 2014. Toward Practical Homomorphic Evaluation of Block Ciphers Using Prince. In *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8438)*, Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith (Eds.). Springer, 208–220. https://doi.org/10.1007/978-3-662-44774-1_17

[49] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9056)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24

[50] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2012), 144. http://eprint.iacr.org/2012/144

[51] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *The 29th IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 01, 2023*. IEEE.

[52] Stephane Foldes. 1980. Symmetries of directed graphs and the Chinese remainder theorem. *J. Comb. Theory, Ser. B* 28, 1 (1980), 18–25. https://doi.org/10.1016/0095-8956(80)90053-2

[53] W. Morven Gentleman and G. Sande. 1966. Fast Fourier Transforms: for fun and profit. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA (AFIPS Conference Proceedings, Vol. 29)*. AFIPS / ACM / Spartan Books, Washington D.C., 563–578. https://doi.org/10.1145/1464291.1464352

[54] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, 850–867. https://doi.org/10.1007/978-3-642-32009-5_49

[55] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, 75–92. https://doi.org/10.1007/978-3-642-40041-4_5

[56] Stefan Groth, Jürgen Teich, and Frank Hannig. 2021. Efficient Application of Tensor Core Units for Convolving Images. In *SCOPES '21: 24th International Workshop on Software and Compilers for Embedded Systems, Eindhoven, The Netherlands, November 1 - 2, 2021*, Sander Stuijk (Ed.). ACM, 1–6. https://doi.org/10.1145/3493229.3493305

[57] Amina Guermouche and Anne-Cécile Orgerie. 2022. Thermal design power and vectorized instructions behavior. *Concurr. Comput. Pract. Exp.* 34, 2 (2022). https://doi.org/10.1002/cpe.6261

[58] Shai Halevi and Victor Shoup. 2021. Bootstrapping for HElib. *J. Cryptol.* 34, 1 (2021), 7. https://doi.org/10.1007/s00145-020-09368-7

[59] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic Regression on Homomorphic Encrypted Data at Scale. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 9466–9471. https://doi.org/10.1609/aaai.v33i01.33019466

[60] Kyoohyung Han and Dohyeong Ki. 2020. Better Bootstrapping for Approximate Homomorphic Encryption. In *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12006)*, Stanislaw Jarecki (Ed.). Springer, 364–390. https://doi.org/10.1007/978-3-030-40186-3_16

[61] Éliane Jaulmes and Antoine Joux. 2000. A Chosen-Ciphertext Attack against NTRU. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1880)*, Mihir Bellare (Ed.). Springer, 20–35. https://doi.org/10.1007/3-540-44598-6_2

[62] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 4 (2021), 114–148. https://doi.org/10.46586/tches.v2021.i4.114-148

[63] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. *CoRR* abs/2205.00922 (2022). https://doi.org/10.48550/arXiv.2205.00922 arXiv:2205.00922

[64] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. 2020. Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs. *CoRR* abs/2012.01968 (2020). arXiv:2012.01968 https://arxiv.org/abs/2012.01968

[65] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 711–725. https://doi.org/10.1145/3470496.3527415

[66] Ian Kuon and Jonathan Rose. 2006. Measuring the gap between FPGAs and ASICs. In *Proceedings of the ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, FPGA 2006, Monterey, California, USA, February 22-24, 2006*, Steven J. E. Wilton and André DeHon (Eds.). ACM, 21–30. https://doi.org/10.1145/1117201.1117205

[67] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. *IEEE Access* 10 (2022), 30039–30054. https://doi.org/10.1109/ACCESS.2022.3159694

[68] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2022. INSPIRE: in-storage private information retrieval via protocol and architecture co-design. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 102–115. https://doi.org/10.1145/3470496.3527433

[69] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. 2021. PEGASUS: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1057–1073. https://doi.org/10.1109/SP40001.2021.00043

[70] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6110)*, Henri Gilbert (Ed.). Springer, 1–23. https://doi.org/10.1007/978-3-642-13190-5_1

[71] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. 2020. Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme. *IEEE Trans. Very Large Scale Integr. Syst.* 28, 2 (2020), 353–362. https://doi.org/10.1109/TVLSI.2019.2943127

[72] Vincent Migliore, Cédric Seguin, Maria Mendez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat, and Russell Tessier. 2017. A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 138:1–138:17. https://doi.org/10.1145/3126558

[73] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 19–38. https://doi.org/10.1109/SP.2017.12

[74] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2292–2306. https://doi.org/10.1145/3460120.3485381

[75] Erdinç Öztürk, Yarkin Doröz, Erkay Savas, and Berk Sunar. 2017. A Custom Accelerator for Homomorphic Encryption Applications. *IEEE Trans. Computers* 66, 1 (2017), 3–16. https://doi.org/10.1109/TC.2016.2574340

[76] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding (Lecture Notes in Computer Science, Vol. 1592)*, Jacques Stern (Ed.). Springer, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[77] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 26–39. https://doi.org/10.1109/HPCA51647.2021.00013

[78] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1295–1309. https://doi.org/10.1145/3373376.3378523

[79] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126. https://doi.org/10.1145/359340.359342

[80] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 387–398. https://doi.org/10.1109/HPCA.2019.00052

[81] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 238–252. https://doi.org/10.1145/3466752.3480070

[82] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 173–187. https://doi.org/10.1145/3470496.3527393

[83] Nigel P. Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.* 71, 1 (2014), 57–81. https://doi.org/10.1007/s10623-012-9720-4

[84] H. Tian, C. Zeng, Z. Ren, D. Chai, J. Zhang, K. Chen, and Q. Yang. 2022. Sphinx: Enabling Privacy-Preserving Online Learning over the Cloud. In *2022 IEEE Symposium on Security and Privacy, SP 2022, Los Alamitos, CA, USA, May, 2022*. IEEE Computer Society, 1135–1149. https://doi.org/10.1109/SP46214.2022.00066

[85] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA. *IEEE Trans. Computers* 69, 8 (2020), 1185–1196. https://doi.org/10.1109/TC.2020.2988765

[86] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. 2021. SoK: Fully Homomorphic Encryption Compilers. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1092–1108. https://doi.org/10.1109/SP40001.2021.00068

[87] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM Trans. Intell. Syst. Technol.* 10, 2 (2019), 12:1–12:19. https://doi.org/10.1145/3298981

[88] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *The 29th IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 01, 2023*. IEEE.

[89] Yilan Zhu, Xinyao Wang, Lei Ju, and Shanqing Guo. 2023. FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference. In *The 29th IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 01, 2023*. IEEE.

A NTT/FFT

Given a polynomial A of degree n as follows:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \tag{3}$$

The polynomial can be represented in the form of the vector of its coefficients as $\boldsymbol{a} = (a_1, a_2, ..., a_{n-1})$. While the addition operation of two polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ (*i.e.*, element-wise addition of their coefficients vectors) is trivial, the multiplication of $\boldsymbol{a}$ and $\boldsymbol{b}$ (denoted as $\boldsymbol{a} \bigotimes \boldsymbol{b}$ in our paper) is time-consuming with the computation complexity of $O(n^2)$. Since polynomial multiplication is the fundamental operation in FHE schemes, *e.g.*, encryption, homomorphic operations, *etc.*, such high computation complexity causes FHE schemes to be extremely inefficient.

To optimize polynomial multiplication, another representation of the polynomial — point-value representation — can be exploited. A polynomial of degree n can be represented by n distinct point-value tuples: $\{(x_0, y_0), (x_1, y_1), ...(x_{n-1}, y_{n-1})\}$, where $y_k = A(x_k)$ and $k \in [0, n)$. To achieve calculations between polynomials, *e.g.*, A and B, the same group of x_k is chosen from both polynomials A and B. Under point-value representation, both addition and multiplication between two polynomials of degree n only require n point-wise operations on y_k. Therefore, the computation complexity is reduced to $O(n)$.

However, the conversion from coefficient representation to point-value representation is still time-consuming and requires the calculation of $y_k = A(x_k)$ for every x_k. The time complexity of evaluating each $A(x_k)$ is $O(n)$, thus leading to overall time complexity of $O(n^2)$ for $k \in [0, n)$. The same complexity is needed for the inverse conversion.

To lower the time complexity of representation conversion, we choose special x_k values. For a polynomial of degree $n-1$, we use the roots of an equation $\omega^n = 1$ as x_k values. The equation has n roots, denoted as $e^{2\pi ik/n}$, $k = 0, 1, ..., n-1$. They are usually written in simplified form as $\omega_n^k = e^{2\pi ik/n}$, which is the power of $\omega_n = e^{2\pi i/n}$. With these roots, we can write Equation 3 as

$$A_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}, k \in [0, n) \tag{4}$$

Equation 4 is called Discrete Fourier Transform (DFT). Our goal is to accelerate the transformation with the property of ω_n.

For ω_n, we have the following theorems:

$$\omega_{2n}^{2k} = \omega_n^k \tag{5}$$

$$\omega_n^{k+\frac{n}{2}} = -\omega_n^k \tag{6}$$

Given a polynomial $A(x) = a_0 + a_1 \cdot x^1 + a_2 \cdot x^2 + ... + a_{n-1} \cdot x^{n-1}$, we can re-arrange it to $A(x) = (a_0 + a_2 \cdot x^2 + a_4 \cdot x^4 + ... + a_{n-2} \cdot x^{n-2}) + x(a_1 + a_3 \cdot x^2 + a_5 \cdot x^4 + ... + a_{n-1} \cdot x^{n-2})$. Let $A^{[0]}(x) = a_0 + a_2 \cdot x^1 + a_4 \cdot x^2 + ... + a_{n-2} \cdot x^{\frac{n-2}{2}}$ and $A^{[1]}(x) = a_1 + a_3 \cdot x^1 + a_5 \cdot x^2 + ... + a_{n-1} \cdot x^{\frac{n-2}{2}}$. Then we have $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$.

Therefore, we can use $x_k = \omega_n^k$ to sample values. For $k = 0, 1, \cdots, n/2 - 1$, we have

$$A(\omega_n^k) = A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \tag{7}$$

Based on Theorems 5, the equation transforms into

$$A(\omega_n^k) = A^{[0]}(\omega_{\frac{n}{2}}^k) + \omega_n^k A^{[1]}(\omega_{\frac{n}{2}}^k) \tag{8}$$

Similarly, for $k + n/2$, we have:

$$A(\omega_n^{k+\frac{n}{2}}) = A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_n^{2k+n}) \tag{9}$$

Based on Theorems 5 and 6, the equation now becomes:

$$A(\omega_n^{k+\frac{n}{2}}) = A^{[0]}(\omega_{\frac{n}{2}}^k) - \omega_n^k A^{[1]}(\omega_{\frac{n}{2}}^k) \tag{10}$$

Note that k and $k + n/2$ have covered all integers ranging from 0 to $n - 1$. Then the problem becomes a divide-and-conquer problem. The origin problem A is split into two subproblems of $A^{[0]}$ and $A^{[1]}$ whose degrees of polynomial are $n/2$. It requires $n/2$ extra multiplications to recover A from $A^{[0]}$ and $A^{[1]}$. After that, $A^{[0]}$ and $A^{[1]}$ can be further divided into smaller subproblems following the same scheme. The recursion ends when the degree of the polynomial in the subproblem is 1. According to the master theorem in [41], the time complexity can be reduced to $O(n \log n)$.

The aforementioned process is called Fast Fourier Transform (FFT). When the degree of polynomials is n, we refer to the transform as an n-point FFT. The inverse operation of FFT is called iFFT, which can be implemented with a similar approach.

Number Theoretic Transforms (NTT) is similar to DFT but works with the finite field. Different from DFT in Equation 4, NTT is formulated as

$$A_k = \sum_{j=0}^{n-1} a_j g_n^{kj} \mod p, k \in [0, n), \tag{11}$$

where p is a prime, and g_n is the primitive n-th root of unity modulo p, which satisfies the following theorems:

$$g_{2n}^{2k} \equiv g_n^k \mod p \tag{12}$$

$$g_n^{k+\frac{n}{2}} \equiv -g_n^k \mod p \tag{13}$$

With Theorems 12 and 13, NTT can be accelerated following the same scheme we discussed in FFT, except for the multiplication and addition/subtraction, which need to be replaced by modular multiplication and modular addition/-subtraction, respectively.

B 4-step FFT/NTT Algorithm

B.1 Introduction to 4-step FFT/NTT Algorithm

In this section, we use the example of an n-point FFT to illustrate the workflow of the 4-step FFT/NTT algorithm. Recall the original FFT (DFT) in Equation 4. In FHE schemes, n is a power of two and can be expressed as the product of two numbers $n = R \cdot C$. Based on this property, we consider the n input numbers $\{a_j, j \in [0, n)\}$ as an $R \times C$ matrix. In this way, the workflow of the 4-step FFT is as follows.

1. Transpose the $R \times C$ input matrix and get a new $C \times R$ matrix. Perform FFT on each row of the $C \times R$ matrix (*i.e.*, C independent R-point FFTs). We let matrix $\boldsymbol{A'}$ denote the results of FFTs.
2. Transpose matrix $\boldsymbol{A'}$ and get a new $R \times C$ matrix $\boldsymbol{A''}$.

3. Generate a $R \times C$ twisting factor matrix $\boldsymbol{F} = [F_{i,j}]$, where $F_{i,j} = \omega_n^{ij}$. Then perform dyadic multiplication between matrix $\boldsymbol{A''}$ and matrix $\boldsymbol{F}$. Let matrix $\boldsymbol{A'''}$ denote the multiplication result.
4. Perform FFT on each row of the $R \times C$ matrix $\boldsymbol{A'''}$ (*i.e.*, R independent C-point FFTs). Transpose the result of FFT and get a new $C \times R$ matrix, denoted as $\boldsymbol{A}$.

$\boldsymbol{A}$ is the final result of the n-point FFT with some differences in the placement order of the result numbers from the original FFT algorithm. The 4-step NTT algorithm follows a similar workflow with modular operations between integers. Since the order of coefficients does not affect the correctness of element-wise polynomial addition and multiplication, the 4-step FFT/NTT algorithm can be easily applied in FHE schemes to replace the original FFT/NTT algorithm. Actually, we could skip the transpose in the first step by storing the input numbers in column-major order, and skip the transpose in the fourth step because of the order-independent element-wise operations.

B.2 Comparison between the 4-step FFT/NTT Algorithm and the Original FFT/NTT Algorithm

This section will show the pros and cons of the 4-step FFT/NTT algorithm by comparing it with the original FFT/NTT algorithm.

B.2.1 Efficient and Practical Parallelism As mentioned in §4.1.1, the original FFT/NTT algorithm can hardly be accelerated via parallelism because of strict data dependency and high computation resource consumption. The 4-step FFT/NTT algorithm alleviates both problems.

Less Data Dependency: According to the aforementioned workflow, the 4-step FFT/NTT algorithm decomposes the n-point FFT/NTT operation into C independent R-point FFT/NTTs in the first step, a global transpose in the second step, n independent multiplications in the third step, and R independent C-point FFT/NTTs in the last step. Although the four steps should be executed one by one, the parallelism in these steps (except for the second step) is fully exploited as these independent operations can be performed simultaneously. Since R and C can be quite large (*e.g.*, when $n = 65536$, $R = C = 256$), the parallelism is high enough to fully utilize the performance capacity of hardware accelerators. Actually, the global data dependency problem in the original FFT/NTT algorithm still exists in the second step when transposing the matrix. Consequently, a large transpose network is required. Recent works prefer such a design since 1) the transpose network is much easier to design than FFT/NTT due to its fixed workflow; 2) the transpose network can be reused in other operations such as permutation [81], which improves the resource utilization.

Low Resource Consumption: Compared to the original FFT/NTT, the total computation workload in the 4-step FFT/NTT is not reduced. But the original algorithm is decomposed into multiple smaller ones with much fewer input numbers (we choose R and C close to $\sqrt{n}$ in many cases). Therefore, instead of designing a fully pipelined n-point FFT/NTT circuit, we can combine multiple small pipelined FFT/NTT units (*i.e.*, R-point and C-point FFT/NTT units) to achieve the same arithmetic function, reducing the hardware design complexity. To reduce hardware resource consumption, we can reuse R-point FFT/NTT units rather than implement all the units as dedicated ones (*i.e.*, R C-point FFT/NTT units and C R-point FFT/NTT units). Therefore, the 4-step FFT/NTT algorithm requires much fewer hardware resources to achieve pipeline parallelism.

B.2.2 Increased Memory Overhead Although promising, the 4-step FFT/NTT also causes increased memory overhead due to pre-computed parameters. In addition to the input data and calculation results, we need to store the twiddle factors for n-point FFT/NTT in the original algorithm. Larger n leads to more twiddle factors. Although the number of twiddle factors is reduced in the 4-step FFT/NTT algorithm since we only need twiddle factors for R-point and C-point

FFT/NTTs, we need to store the twisting factors introduced in the third step of the 4-step FFT/NTT algorithm. The size of twisting factors in the 4-step FFT/NTT algorithm is close to the size of twiddle factors in the original algorithm, making the requirements of total storage even more strict. To address the problem, recent works generate twiddle and twisting factors on the fly rather than cache all the factors in the memory [65, 82].