UFC: A Unified Accelerator for Fully Homomorphic Encryption

Minxuan Zhou*, Yujin Nam†, Xuan Wang†,Youhak Lee†, Chris Wilkerson‡, Raghavan Kumar‡
Sachin Taneja‡, Sanu Mathew‡, Rosario Cammarota‡, and Tajana Rosing†
*Illinois Institute of Technology †University of California San Diego ‡Intel Labs
Email: mzhou26@iit.edu, {yujinnam, xuw009, yhl004, tajana}@ucsd.edu
{chris.wilkerson, raghavan.kumar, sachin.taneja, sanu.k.mathew, rosario.cammarota}@intel.com

Abstract*—Fully homomorphic encryption (FHE) is crucial for post-quantum privacy-preserving computing. Researchers have proposed various FHE schemes that excel at different encrypted computations, such as single-instruction multiple-data (SIMD) arithmetic or arbitrary single-data functions. Hybrid-scheme FHE, which exploits appropriate schemes for specific tasks, is essential for real-world applications requiring optimal performance and accuracy. However, existing FHE accelerators only adopt scheme-specific custom designs, leading to inefficiency or lack of capability to support applications in hybrid FHE settings. In this work, we propose a *U*nified *F*HE a*C*celerator (UFC) that provides better performance and cost-efficiency than prior scheme-specific accelerators on hybrid FHE applications. Our design process involves a comprehensive analysis of processing flows to abstract the primitives covering all operations in hybrid FHE applications. The UFC architecture primarily comprises hardware function units for these primitives, diverging from the deeply pipelined units in previous designs. This approach enables high hardware utilization across different FHE schemes. Furthermore, we propose several algorithm-hardware co-optimizations to minimize the hardware cost of supporting various data shuffling patterns in FHE. This enables high-throughput implementation of function units that provide good cost efficiency. We also propose several compiler-level optimizations to achieve high hardware utilization of the unified architecture for computing FHE data in various algorithmic parameter settings. We evaluate the performance of UFC on different FHE programs, including scheme-specific and hybrid-scheme workloads. Our experiments show that UFC provides up to 6.0× speedup and 1.6× delay-energy-area efficiency improvement over state-of-the-art FHE accelerators.

***Index Terms*—fully homomorphic encryption, cryptography, hardware acceleration, domain-specific architectures.**

I. Introduction

The big data era has led to a surge in cloud-based outsourcing, raising significant concerns about the security of sensitive data. Fully homomorphic encryption (FHE) offers a promising solution by enabling computations on encrypted data, allowing cloud services to process information without accessing the underlying user data [1], [6]–[11], [16], [19], [24], [35], [45].

Researchers have proposed various FHE schemes, which can be categorized into two main types: 1) schemes that support single-instruction multiple-data (SIMD) arithmetic, such as CKKS [9], BGV [7], and BFV [17] and 2) schemes that support computation on arbitrary logic, such as TFHE [10], [35] and FHEW [16], [36]. Each category works well for specific application scenarios. On the one hand, SIMD schemes [7], [9], [17] are well-suited for high-throughput processing for arithmetic-intensive operations, such as matrix multiplication, convolution, etc. However, these schemes require arithmetic approximation for non-linear functions, necessitating algorithm redesign for accuracy-sensitive algorithms (e.g., machine learning). On the other hand, logic schemes [10], [16], [35], [36] support arbitrary functions on encrypted data. However, each ciphertext only encrypts a single value, resulting in limited throughput. Due to the distinct characteristics of these two categories, recent research has focused on hybrid FHE applications using both schemes with efficient scheme-switching [4], [5], [33]. This hybrid approach enables high-throughput and accurate processing of arbitrary functions, which is not achievable using either logic or SIMD scheme alone.

While algorithm advancements empower FHE to support various applications, hardware acceleration is critical for enabling FHE in real systems because FHE is several orders of magnitude slower and memory-consuming than computation on the original data. Such inefficiency necessitates customized hardware accelerators, which provide more than 10,000× speedup over conventional architectures [23], [26]–[28], [41], [44]. However, existing accelerators are typically scheme-specific, with hardware components optimized for key operations in one FHE scheme category. Specifically, SHARP [26], ARK [27], CraterLake [44], and BTS [28] are customized ASIC accelerators for CKKS [9], which is a popular SIMD scheme. Strix [41], MATCHA [23], and FPT [48] are customized for TFHE [10], a widely used logic scheme. The scheme-specific nature of these accelerators presents a significant limitation: they are not designed to support hybrid FHE programs that utilize multiple FHE schemes. This oversight is problematic for applications requiring both high-throughput computation and accurate results [2], [4], [5], [33].

Despite sharing the same theoretical foundation, SIMD and logic schemes have significant differences in their algorithms and data sizes. Consequently, architecture customization for one scheme often leads to 1) under-utilization when processing the other scheme and 2), in some cases, a complete lack of hardware capability to support the other scheme. For example, SIMD schemes usually adopt large ring learning-with-error (RLWE) [34] polynomials with $logN > 15$ and $logQ > 1000$ to guarantee consecutive homomorphic multiplications on packed data, where N and Q denote polynomial degree and coefficients' bit precision respectively. On the contrary, logic schemes require smaller N and Q, where $logN$ ranges from 10 to 14 and Q is usually less than a word size. In this case, prior SIMD-scheme accelerators [26], [27], [44], which are customized for large polynomials ($logN = 16$, $logQ > 1000$), achieve low hardware utilization for logic-scheme polynomial operations; prior logic-scheme accelerators [23], [41], [48] cannot support SIMD-scheme operations which process data not fit in the hardware (Section III).

This work proposes a new accelerator that can support various FHE schemes in a single hardware while still providing higher performance and cost-efficiency than prior scheme-specific accelerators. Rather than implementing deep pipelines for high-level kernels, our design leverages primitive function units that encompass all operations required by different FHE schemes. This approach enables high hardware utilization across various scenarios. To recognize the key primitives, we first investigate the full spectrum of operations in hybrid FHE applications, including scheme-specific operations and scheme-switching operations. Our investigation found most operations in both schemes share the same underlying arithmetic primitives, showing it is possible for a unified design.

However, scaling out these primitive function units for high-throughput is challenging due to the complex data communication pattern, which mandates expensive on-chip networks. We exploit several algorithm-hardware co-optimizations to mitigate the hardware cost, including the constant-geometry NTT algorithm and communication-free data shuffling operations. These optimizations significantly reduce the hardware cost when increasing the size of primitive function units, enabling high-throughput design with a reasonable chip area. Another challenge of the unified accelerator is to keep high utilization for operation on small data structures (i.e., polynomials), which may not utilize all available hardware resources. To tackle this challenge, we propose several compiler-level scheduling techniques that batch operations on small data structures and optimize the data layout in the memory.

We summarize the key contributions of this work as follows:

- We propose a unified accelerator that accelerates hybrid FHE applications. We exploit several algorithm-hardware co-design techniques to minimize the hardware cost, making it possible to achieve high throughput without deep specialization in hardware.
- We devise several compiler-level optimizations that efficiently utilize the hardware for various FHE operations.
- We conduct a comprehensive design space exploration on the architectural parameters of the proposed accelerator by evaluating FHE workloads in various scenarios, including scheme-specific workloads and hybrid workloads.
- Experiments show that UFC provides superior performance and energy-delay-area-product (EDAP) to prior scheme-specific accelerators. Specifically, UFC provides up to 6.0× speedup and 1.6× better EDAP over prior scheme-specific accelerators [26], [41] on scheme-specific workloads and a composed system with multiple accelerators on hybrid workloads.

II. Background

We first introduce the basics of FHE, using the terminology based on previous works [26], [41]. Then, we describe the high-level operations for various FHE schemes and scheme-switching techniques used in hybrid FHE applications.

A. FHE Basics

Most commonly used FHE schemes nowadays are based on the (ring) learning with errors (LWE) problem [34]. The ciphertext types used in the schemes include LWE, ring LWE (RLWE), and RGSW.

1) LWE: For dimension n and modulus q, an LWE encryption of a message $m \in \mathbb{Z}$ is $LWE_{\vec{s}}(m) = (\vec{a}, b)$, where $\vec{a} \leftarrow \mathbb{Z}_q^n$ and $b = -\vec{a} \cdot \vec{s} + m + e \in \mathbb{Z}_q$. Secret key $\vec{s}$ is a vector $(s_0, s_1 ... s_{n-1})$, sampled from a distribution χ.

2) RLWE: Ring LWE is defined over a ring $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$, where $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, q is ciphertext modulus and N is ring dimension. Secret key $\mathbf{s}$ is sampled from a polynomial ring $\mathcal{R}_q$. An RLWE encryption is $RLWE_{\mathbf{s}}(\mathbf{m}) = (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_{\mathbf{q}}^{\mathbf{2}}$. $\mathbf{a} \leftarrow \mathcal{R}_{\mathbf{q}}$ and $\mathbf{b} = -\mathbf{a} \cdot \mathbf{s} + \mathbf{m} + \mathbf{e}$.

3) RGSW: RGSW follows the same secret key and parameters of RLWE and can be understood as a matrix of RLWE encryptions generated by gadget decomposition. For a gadget vector $\vec{g} = (g_0, g_1 ... g_{k-1})$, we define a gadget matrix $\mathbf{G} = \mathbf{I}_2 \otimes \vec{g}$. $RGSW_{\mathbf{s}}(\mathbf{m})$ equals to $\mathbf{Z} + \mathbf{G} \cdot \mathbf{m} \in \mathcal{R}_q^{2k \times 2}$. Each row of $\mathbf{Z}$ is an RLWE encryption of a zero, $RLWE_{\mathbf{s}}(\mathbf{0})$.

4) Number theoretic transform: Number theoretic transform (NTT) algorithm, or discrete Fourier transform (DFT) algorithm over finite field, is a key algorithm for FHE. NTT is used to evaluate the multiplication over polynomial ring $\mathcal{R}_q$. This is because a polynomial multiplication, or a negacyclic convolution in non-NTT domain equals to an element-wise multiplication in the NTT domain: $\mathbf{c} = iNTT(NTT(\mathbf{a}) \circ NTT(\mathbf{b}))$. NTT operation uses n-th primitive root of unity ω, such that $\omega^n \equiv 1 \mod q$. For an input $\mathbf{a} = (\mathbf{a_0}, \mathbf{a_1} \dots \mathbf{a_{N-1}})$, NTT algorithm evaluates $A_k = \sum_{n=0}^{N-1} a_n \omega^{nk} \mod q$ for $k = 0, 1, \dots N-1$. For more details on NTT algorithms, we refer the readers to a recent survey [31].

5) FHE schemes: Since the ground-breaking work by Gentry [19], researchers have proposed various FHE schemes that exploit different algorithms to encrypt the data and support homomorphic operations [6], [7], [9], [10], [16], [17], [35], [36], illustrated in the following sections.

B. SIMD FHE Schemes

The SIMD FHE schemes only adopt RLWE ciphertexts as the building block while choosing large parameter settings to encrypt a vector of values into a single RLWE ciphertext, $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}_{\mathbf{q}}^{\mathbf{2}}$ [6], [7], [9], [17]. By exploiting the isomorphism, operations on a single ciphertext can homomorphically manipulate all values in the plaintext vector. Such schemes support homomorphic operations on element-wise arithmetic and arbitrary vector rotation. The representative SIMD schemes include BGV [7], BFV [17], and CKKS [9]. This work uses

CKKS as the basic SIMD scheme because of its support for fast processing on real numbers, which is suitable for emerging applications like neural networks [29].

1) Homomorphic operations: CKKS supports homomorphic addition and multiplication between two ciphertexts or one ciphertext and one plaintext. Homomorphic multiplication between two ciphertexts, $(\mathbf{a_0}, \mathbf{b_0}) \in \mathcal{R}_\mathbf{q}^\mathbf{2}$ and $(\mathbf{a_1}, \mathbf{b_1}) \in \mathcal{R}_\mathbf{q}^\mathbf{2}$, is complex because the result is a ciphertext $(\mathbf{a_0 a_1}, \mathbf{a_0 b_1} + \mathbf{a_1 b_0}, \mathbf{b_0 b_1}) \in \mathcal{R}_\mathbf{q}^\mathbf{3}$. To transform the result to $\mathcal{R}_q^2$, CKKS requires a key-switching process that multiplies the resulting ciphertext with a special ciphertext, key switching key (ksk). Due to the significant noise growth after homomorphic multiplication, a rescaling operation is adopted to divide the ciphertext by q_{rs}. The ciphertext modulus after rescaling becomes q/q_{rs}. Therefore, CKKS can only support a limited number of continuous multiplications.

CKKS also supports homomorphic r-rotation by applying an *automorphism*, which maps i-th coefficient of polynomials in the ciphertext to $(i \cdot 5^r mod\ N)$-th coefficient. Similar to homomorphic multiplication, homomorphic rotation is also followed by a key-switching process.

2) Residual number system (RNS): Due to the large coefficient size, the latest CKKS scheme adopts the residual number system (RNS) to decompose polynomials into multiple RNS polynomials, each having word-size coefficients. For polynomials in R_Q, the scheme chooses a set of pair-wise coprime integers q_i where $i \in [0, L)$ and $q_0 q_1 ... q_L = Q$. Each polynomial a is represented by L polynomials $a[0...L]$, where $a[i] \in R_{q_i}$. These RNS moduli are also the leveled modulus, so each rescaling removes one RNS polynomial in the ciphertext.

3) Key-switching: Key switching is the most expensive high-level operation in CKKS. Its key is the multiplication between the input ciphertext c and the evaluation key ksk. To avoid overflow on modulus $Q = q_0 q_1 ... qL$, ksk has a larger modulus PQ with the special modulus $P = p_0 p_1 ... p_K$. Thus the first step is to convert c with modulus Q into a ciphertext with PQ by a base conversion $BConv_{Q,P}(a_Q) = ([\sum_{j=0}^{L} [a[j] * \hat{q_j}^{-1}]_{q_j} * \hat{q_j}]_{p_i})_{0 \le i < K}$.

4) Bootstrapping: Bootstrapping is a technique to recover the modulus of ciphertext back to a larger size to support an arbitrary number of homomorphic computations [21]. The CKKS bootstrapping is complex, requiring homomorphic linear transformations and approximation of floor function using homomorphic operations.

C. Logic FHE Scheme

The logic-based schemes use LWE as the basic encryption format [10], [16]. However, they also use RLWE and RGSW internally for different operations. This work focuses on TFHE [10], a widely used efficient logic-based scheme.

1) Homomorphic operations: LWE ciphertext supports a few trivial operations, including addition and scalar multiplication. For example, addition is an element-wise addition between two LWEs, and scalar multiplication can be done by multiplying every element with a scalar value. These functionalities can be shown trivially with the definition of LWE from section II-A. However, other operations require more complicated procedures with functional bootstrapping.

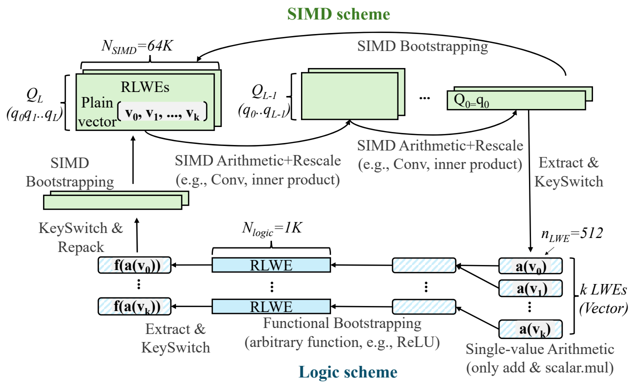


Fig. 1. The overview of the hybrid FHE programs with two schemes.

2) Functional bootstrapping: Functional bootstrapping homomorphically computes a function $f(\cdot)$ over an encryption of m during bootstrapping. The high-level procedure of functional bootstrapping follows three steps: packing, accumulation, and extraction. [10], [41] The packing step packs input LWEs containing function f's values into an RLWE to initialize the test vector tv. This step includes the modulus switching from LWE modulus q to RLWE modulus q'. The accumulation, or a homomorphic decryption step, uses the bootstrapping key $\mathbf{btk}_i$, $i \in [n]$, which is an RGSW encryption of the secret key $\mathbf{s}$, under another secret key $\mathbf{s}'$. The step iteratively evaluates the external product between tv and $\mathbf{btk}_i$. The tv here needs to go through NTT and iNTT operations before each external product, making this step one of the most expensive steps of functional bootstrapping. Finally, we extract an LWE from the RLWE result from the accumulation step (i.e., tv). Such LWE is still encrypted under $\mathbf{s}'$, which needs a key-switching procedure to switch back to the original key $\mathbf{s}$.

3) Key-switching: Key-switching in TFHE converts an LWE encryption under key $\mathbf{s}'$ into an LWE encryption of the same message under another key $\mathbf{s}$. [10], [41] This procedure uses key switching key $\mathbf{ksk}_{i,j}$, where $i \in [n], j \in [d_{ks} = \lceil \log_{B_{ks}} q \rceil]$. The key $\mathbf{ksk}_{i,j}$ is a set of LWEs parameterized by the key $\mathbf{s}$'s dimension n and key decomposition base B_{ks}. Each element of the input ciphertext $LWE_\mathbf{s}(m)$ goes through a base expansion according to the base B_{ks}, which is followed by the multiplication with the key $\mathbf{ksk}_{i,\cdot}$.

D. Hybrid FHE Programs with Scheme-Switching

Both SIMD schemes and logic schemes are based on (Ring)LWE problems, and it is possible to convert ciphertexts between schemes. Recent research also shows the significant advantage of utilizing a hybrid scheme to support high-throughput arithmetic and precise computation simultaneously [2], [4], [5], [33]. In addition to scheme-specific operations, hybrid FHE programs require scheme-switching operations to convert ciphertext in one scheme to another.

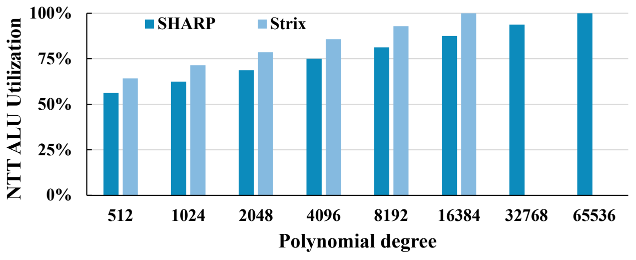


Fig. 2. Hardware utilization of NTT unit on SHARP [26] and Strix [41] for polynomials with different degrees.

Figure 1 shows the overall flow of hybrid FHE programs with hybrid schemes. The technique that converts LWE ciphertext (logic-based scheme) to RLWE ciphertext (SIMD scheme) is *repacking*. *Repacking* can convert multiple LWE ciphertexts into one RLWE ciphertext. The recent *repacking* algorithm uses homomorphic linear transformation followed by a key switching and a bootstrapping in SIMD schemes [33]. The technique of converting an RLWE ciphertext to one or more LWE ciphertexts is called *extraction*, the same as the *extraction* used in the functional bootstrapping of logic-based schemes. Due to the discrepancy of algorithmic parameters, the *extraction* requires a TFHE key-switching at the end to convert the extracted LWE ciphertexts back to the standard parameter setting for logic-based homomorphic operations.

III. Motivation: Hybrid FHE Programs on Existing Accelerators

Hardware acceleration is critical to enabling FHE in the real world because FHE is exceptionally inefficient on conventional systems. However, most existing FHE accelerators adopt deeply customized function units to optimize the system throughput for a specific scheme [23], [26], [28], [41], [43], [44]. Unfortunately, such scheme-specific designs cannot support hybrid FHE applications efficiently due to the incompatibility of algorithmic parameters and the lack of capability to support non-considered schemes.

A. Accelerators for SIMD Schemes

SIMD FHE schemes require large parameter settings in the FHE algorithms to efficiently support NTT and RNS-related operations (e.g., BConv). Existing SIMD-scheme accelerators utilize the deeply pipelined function units for NTT on large polynomials and multiply-and-accumulations (MACs) in BConv. However, these customized NTT units adopt scheme-specific optimizations in the hardware design. Specifically, NTT units used by prior SIMD-scheme accelerators [26], [44] adopt a multi-stage pipeline for polynomials with $logN = 16$, where each stage supports $\sqrt{N}$ throughput with transposition in the middle of the pipeline. Even though such a design can support smaller polynomials of logic schemes by bypassing some stages, the hardware utilization decreases with the polynomial size, leading to low performance-area cost-efficiency. Figure 2 shows the hardware utilization for different polynomials degrees on prior SIMD-scheme accelerator (SHARP [26]), showing 50% to 75% utilization for commonly used settings in logic schemes (i.e., $logN = 9 - 12$).

In addition to the unstable NTTU utilization, the customized pipelines for BConv in prior SIMD-scheme accelerators duplicate the hardware resource to support parallel MACs from one input RNS polynomial for a batch of target RNS polynomials (up to 60 [44]). However, logic-FHE schemes require a completely different MAC pattern, which happens during the decomposed multiplication. When processing MACs in logic schemes, BConv units in SIMD-scheme accelerators can only activate 1 lane, making most hardware idle. Such results show that existing SIMD-scheme accelerators only unleash a small portion of hardware performance for logic schemes. In addition to the utilization issues, existing SIMD-scheme accelerators cannot support some operations required by logic schemes, such as polynomial rotation, bit decomposition, etc.

B. Accelerators for Logic Schemes

Like SIMD-scheme accelerators, state-of-the-art logic-scheme accelerators [23], [41] also adopt deeply customized pipelines for a specific set of algorithmic parameters and operations. The key function units include FFT units (FFTUs, similar to NTTUs), vector-multiply-add units (MACUs), accumulation units, decomposition units, etc. Due to the customization, these function units cannot support operations in SIMD schemes. Specifically, FFTU in Strix [41] only supports polynomials with $logN \leq 14$, as shown in Figure 2, where the hardware utilization of Strix [41] also decreases with the polynomial degree. Furthermore, other function units, such as MACUs, cannot be reused for similar operations in SIMD schemes because they are integrated with pipelines for high-level routines, including functional bootstrapping and key-switching.

IV. Unified FHE Accelerator

In this work, we design a unified FHE accelerator to support all operations with high hardware utilization for hybrid FHE applications. To achieve these goals, we follow two principles: 1) the accelerator only uses the primitive hardware components that both schemes can fully utilize, and 2) the architecture can be scaled to support high throughput processing. We first conduct a comprehensive investigation of various operations in hybrid FHE programs, which abstracts several general primitive operations that need to be supported in the hardware. Then, we propose a novel FHE accelerator architecture that exploits these function units for general primitive operations as the building block. We also adopt algorithm-hardware co-design techniques to scale the hardware for high throughput without significantly increasing the hardware cost.

A. Operation Breakdown in Hybrid FHE Programs

To design the hardware with high utilization in hybrid FHE applications, we comprehensively investigate all operations to unveil the underlying data flow and computation pattern. Then, we conduct a taxonomy of these operations and abstract a minimum set of function units that cover all required operations in

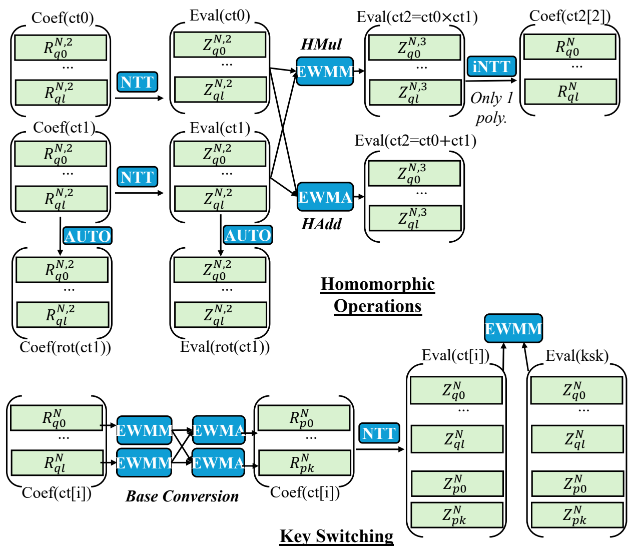


Fig. 3. Operation breakdown of CKKS.

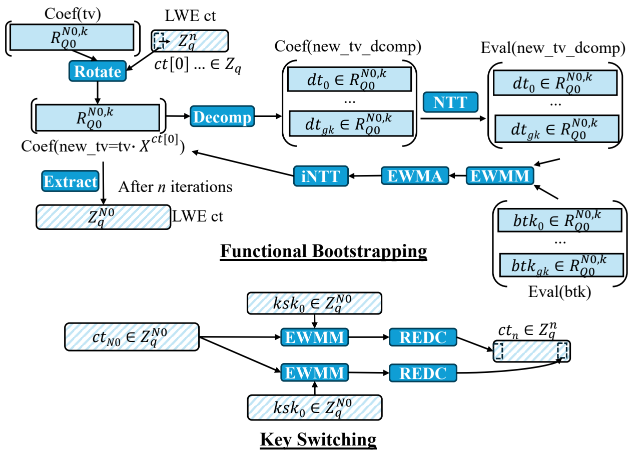


Fig. 4. Operation breakdown of TFHE.

TABLE I
GENERAL FHE PRIMITIVES

	Data	Computation	Shuffle
(i)NTT	RLWE	Parallel butterfly operations	all-to-all
EWMM/A	RLWE/LWE	Parallel modular arithmetic	N/A
AUTO	RLWE	Negate	all-to-all
Rotate	RLWE	Negate	Circular
Extract	RLWE→LWE	Negate	all-to-all
Decomp	RLWE	Bit masking	N/A
REDC	LWE→scalar	Reduction	N/A

the hybrid FHE programs, as shown in Figure 3 and Figure 4 for CKKS and TFHE, respectively. We recognize several primitive operations for each scheme. Specifically, both CKKS and TFHE require (i)NTT, element-wise modular multiplication (EWMM), and element-wise modular addition (EWMA). In addition, CKKS requires automorphism (AUTO) to rotate the

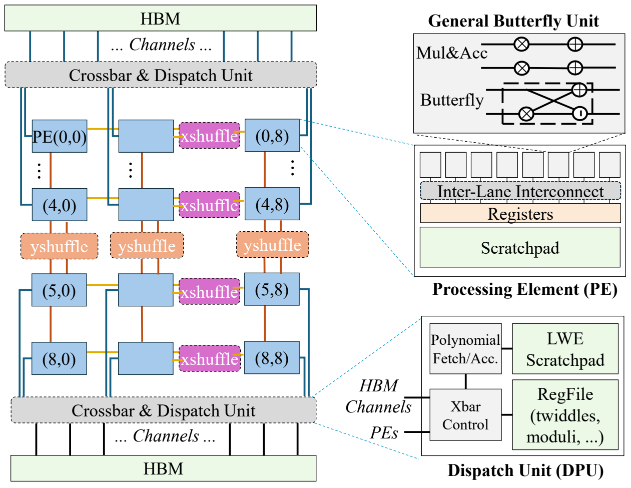


Fig. 5. The UFC architecture.

data homomorphically. TFHE requires more scheme-specific primitives, including coefficient rotation (Rotate), LWE extraction (Extract), vector decomposition (Decomp), and vector reduction (REDC). We summarize the detailed information of these primitives in Table I.

The key observation is that hybrid FHE programs require highly parallel computations, including butterfly operation and modular arithmetic. This indicates we can build a high-throughput architecture with wide function units, along with appropriate interconnect networks for various permutations, for these operations instead of relying on highly customized pipelines in prior FHE accelerators. The key challenge is handling various data shuffling patterns, especially in large-scale, high-throughput designs, because the wiring complexity of various shuffling patterns increases significantly with the number of inputs. We tackle these challenges using several algorithm-hardware co-optimizations, including an interconnect architecture for constant-geometry (i)NTT and shuffle-free algorithms for automorphism and polynomial rotation, in Section IV-C. Furthermore, we adopt a lightweight hardware unit near the memory controller to efficiently handle several non-parallelizable LWE operations, including LWE extraction and vector reduction, in Section IV-B4.

B. UFC Architecture

Figure 5 shows the overall architecture of the proposed unified accelerator, UFC. Based on our operation taxonomy, UFC contains an array of general processing elements (PEs), each only adopts function units for primitive operations. Each function unit has a number of lanes to support high-throughput processing. UFC adopts a general hierarchical memory where each PE has several banked register files and a scratchpad memory. Furthermore, each scratchpad memory is connected to a channel from memory, which is high-bandwidth memory (HBM) in our design. To support various permutation operations, UFC adopts interconnect networks in different hierarchical levels, including the lane level, the PE level, and the channel level in the memory. We exploit several algorithm-hardware co-designs to significantly reduce the cost of interconnect networks, which is essential to ensure the scalability of UFC (Section IV-C). Furthermore, UFC adopts several specialized hardware components to support on-die data generation, data reference, and data reallocation efficiently. We describe the details of each component in the following paragraphs.

1) General Processing Element Array: Unlike existing FHE accelerators that rely on deeply pipelined function units for high throughput, UFC utilizes a flattened architecture that adopts function units for primitive operations on very wide data. Specifically, each processing element has multiple lanes (e.g., 64) for butterfly operations, modulo arithmetic operations (e.g., multiply, add, subtraction, etc.), and digit decomposition. UFC scales the number of general processing elements to support high processing throughput and organizes them in a 2D array to balance the chip dimensions. When processing FHE data, including polynomials in either coefficient or evaluation form, UFC distributes the vector elements to all lanes across all PEs in the system. Each lane will process multiple elements sequentially for vectors with more elements than the total lane counts. If a vector has fewer elements than the total lane counts, UFC only assigns a subset of lanes. In Section V-A, we introduce several compiler-level optimizations to utilize UFC lanes for small data structures efficiently.

2) Memory Hierarchy: UFC adopts various memory components at different levels. Each UFC PE has a register file that stores wide data entries with 1R1W ports. The register file has 4 banks to input data to different functional lanes. Specifically, the butterfly operation uses bank 0 and bank 1, and the element-wise operations will use all 4 banks. Furthermore, each PE register file has sufficient entries to accommodate the partial accumulation results required by base conversion in CKKS and external products in TFHE. In addition to the register file, each PE contains a scratchpad to accommodate more on-chip data. Each scratchpad connects to an HBM channel to fully utilize the memory bandwidth. Vector data is distributed in HBM channels in the same way as the scratchpad. UFC adopts 2 HBM3 PHYs, which support 64 (pseudo)channels in total, resulting in a 64-PE architecture.

3) Interconnect Networks: FHE applications require various data communication patterns between lanes, necessitating corresponding interconnects. We add several interconnect networks at different hierarchical levels. Specifically, we design the inter-lane and inter-PE networks to support constant-geometry NTT algorithms to minimize the required data movement patterns in Section IV-C. Furthermore, we add a crossbar network between HBM channels to support data shuffling required by small polynomial packing (Section V-A). To minimize the hardware cost of interconnect, especially when deploying a large number of lanes/PEs, we propose several algorithm-hardware co-designs introduced in Section IV-C.

4) Near-memory LWE Unit: We propose a dedicated LWE unit (LWEU) near the HBM-channel crossbar to handle various LWE operations, including LWE extraction, coefficient reduction in TFHE key switching, and data preparation for polynomial rotation in functional bootstrapping. Specifically, during LWE extraction, LWEU reads the RLWE ciphertext from the distributed PE scratchpads and reorders the polynomial coefficients in the LWEU scratchpad. Similarly, after TFHE key switching, LWEU reads partial products from PE scratchpads and reduces them into the result LWE ciphertext. During the functional bootstrapping, the polynomial operation (i.e., polynomial rotation) depends on each coefficient in the LWE ciphertext. In this case, LWEU reads coefficients from the LWE scratchpad individually and dispatches corresponding data and instruction (Section IV-C) to PEs.

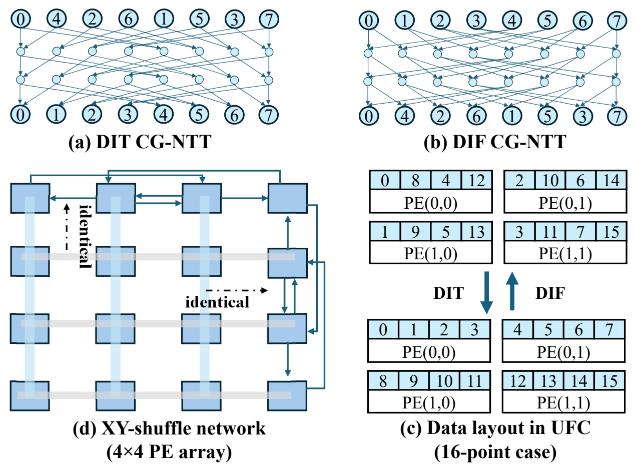


Fig. 6. The constant geometry NTT and its mapping to UFC.

5) On-the-fly Generation for Keys and Twiddles: To reduce the memory bandwidth pressure, we reuse the on-the-fly data generation units from previous works [26], [27], [44], including the key switching key and twiddle factors.

C. Algorithm-hardware co-design for Efficient Interconnect

One major challenge in UFC is the high cost of the interconnect network, which needs to handle all-to-all data communication across lanes to support high-throughput FHE operations with extensively parallel function units for primitive operations. The network complexity significantly increases with the number of processing lanes in UFC. We adopt several algorithm-hardware co-optimizations to minimize the complexity of the interconnect network.

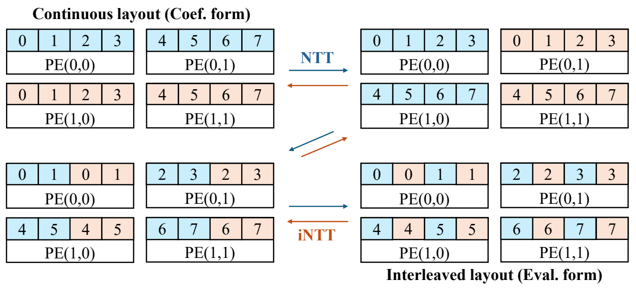
1) Constant-geometry NTT: NTT is the key operation in FHE, which requires $log(N)$ steps of permutations and computations, where N denotes the polynomial size. Prior FHE accelerators exploit the classical NTT algorithm [32], which requires a specific communication pattern for each permutation step. Due to the symmetric structure of these permutation networks, the hardware requires $log(E)$ networks, where E denotes the total lane count in the NTT unit, leading to trade-offs between throughput and hardware cost. Therefore, adopting the classical NTT algorithm in UFC, which contains a large number of lanes, is not practical. To tackle such challenges, we adopt the constant-geometry NTT (CG-NTT) algorithm [40], which is an NTT algorithm requiring only 1 permutation pattern across log(N) steps, as shown in Figure 6(a)(b). We can use the decimation-in-time (DIT) algorithm and decimation-in-frequency (DIF) algorithm for iNTT and NTT, respectively, to minimize the number of interconnections [31], [32]. Figure 6(c) shows the data mapping of a 16-point polynomial to the 2D PE array in UFC. Even though the CG-NTT algorithm minimizes the required interconnections, the hardware cost is still high due to the long wires crossing different PEs. This makes placement and routing difficult. Based on our experiments, the CG-NTT network may increase the chip area of an 8×8 PE array by up to 5×, which is prohibitively expensive.

We further exploit the permutation decomposition technique to reduce the chip area [37]. Based on previous work [37], the constant-geometry permutation can be decomposed into three steps in the 2D PE array, including 1) data shuffling in the x-axis (*xshuffle*), 2) data shuffling in the y-axis (*yshuffle*), and 3) data shuffling within each PE (*rshuffle*). In each shuffling step, all PEs can process data transfer independently. We can use the intra-PE interconnect to combine the *rshuffle* and the butterfly operation and improve the throughput by pipelining three steps in different NTT operations. Even though the decomposition increases the number of wires, it limits all wires in horizontal or vertical directions, leading to a much smaller chip area than directly implementing a constant geometry network.

Even though CG-NTT significantly improves the cost-efficiency of increasing system throughput, it causes a problem with utilizing the large CG-NTT network for small polynomials required by logic-based FHE schemes. In Section V-A, we introduce a compiler-level optimization with the help of an inter-channel crossbar to tackle such challenges efficiently.

2) Automorphism via (i)NTT: In addition to NTT, automorphism is another operation that requires all-to-all data communications across lanes. Automorphism follows the permutation pattern in the polynomial: $X^i \rightarrow X^{ik \ mod \ N}$, where k varies based on the rotation step. Therefore, adding networks for all automorphism patterns is extremely costly in UFC, considering the large number of lanes. We exploit the basic theory of NTT, which transforms the polynomial coefficients to evaluations on ω^0, ω^1, ..., ω^N, where ω is the root of the unit. Therefore, if we replace ω^i with ω^{ik} during the NTT process, the result of NTT will be the evaluation form of $f(X^k)$, the same as the result of automorphism. In this case, we can reuse the NTT interconnect to process any automorphism operations by setting the appropriate root of the unit for computation. Considering automorphism is followed by base conversion for key-switching, requiring polynomial in the coefficient form, the proposed replacement requires extra NTT with ω^{ik} and iNTT with ω.

3) Rotation via Polynomial Multiplication: Another operation requiring data shuffling is the first step of each iteration in functional bootstrapping, where the test vector needs to multiply a polynomial X^{ai}. Instead of handling it as a rotation



Continuous layout (Coef. form)
NTT
iNTT
Interleaved layout (Eval. form)
PE(0,0)
PE(0,1)
PE(1,0)
PE(1,1)

Fig. 7. Packing two small polynomials (red and blue) on UFC.

operation, UFC load the evaluation (NTT) form of X^{ai} and multiply it with the evaluation form of the test ciphertext to avoid extra interconnect networks for costly rotation across lanes. Therefore, the LWEU just needs to dispatch the predefined evaluation form of polynomial X^{ai} to PEs based on the corresponding coefficient in the LWE ciphertext.

V. Scheduling and Data Layout Optimizations

The processing flow, including scheduling and data layout strategy, is critical to fully utilize UFC hardware for FHE operations. Mapping SIMD-FHE operations onto UFC is straightforward because each ciphertext can fully utilize all the lanes in UFC. Unlike SIMD-FHE schemes, logic-FHE schemes work on polynomials that may be smaller than the available throughput in the hardware. Therefore, we propose several optimizations to schedule logic-scheme FHE operation on UFC efficiently.

A. Small Polynomial Packing

The 2D PE array in UFC supports operations on wide data (high-degree polynomial), which might be several times wider than data in a logic-based scheme. Therefore, to achieve full hardware utilization, we need to pack multiple small polynomial operations in UFC. However, this causes a complication in the proposed architecture, which only supports a fixed permutation pattern for the constant geometry NTT.

We tackle such a challenge by allowing two different data layouts in the hardware for polynomials: 1) continuous layout and 2) interleaved layout, as shown in Figure 7. UFC stores all polynomials in the coefficient (original) form in a continuous data layout. We can apply a decimation-in-frequency (DIF) CG-NTT algorithm to such packed small polynomials. The NTT algorithm only runs $log(M)$ instead of $log(N)$ steps, where M and N denote the degree of small polynomial and the number of lanes, respectively. After the DIF CG-NTT on packed polynomials, all polynomials are transformed to the evaluation form, where coefficients of different polynomials are interleaved. On the other hand, we can apply a decimation-in-time (DIT) CG-iNTT algorithm on the interleaved polynomials, which are transformed back to the coefficient form in the continuous layout.

The proposed small polynomial packing technique supports the full hardware utilization for small data structures while

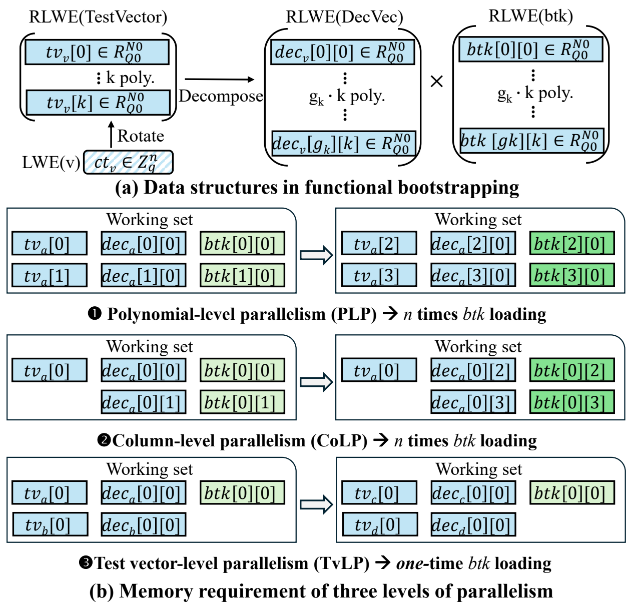

RLWE(TestVector)
RLWE(DecVec)
RLWE(btk)
Decompose
Rotate
LWE(v)
(a) Data structures in functional bootstrapping
Working set
❶ Polynomial-level parallelism (PLP) → n times btk loading
❷Column-level parallelism (CoLP) → n times btk loading
❸Test vector-level parallelism (TvLP) → one-time btk loading
(b) Memory requirement of three levels of parallelism

Fig. 8. Exploiting parallelism for small polynomial packing.

TABLE II
ARCHITECTURE CONFIGURATION OF UFC

Processing Element (PE)	
Butterfly ALU	128
Mod.ADD/Mul	256
Register File	$72 \times 4 \times$ 1KB
Local Interconnect	$256 \times$ 4B
Compute Cluster	
PE	8×8
Global Interconnect	2,048 $\times$ 4B $\times$ 16
Scratchpad	$64 \times$ 4MB
Near-memory Unit	
Crossbar	$32 \times 32 \times 2$
LWE SPAD	32KB
Area/Power@7nm: 197.7mm^2/76.9W	

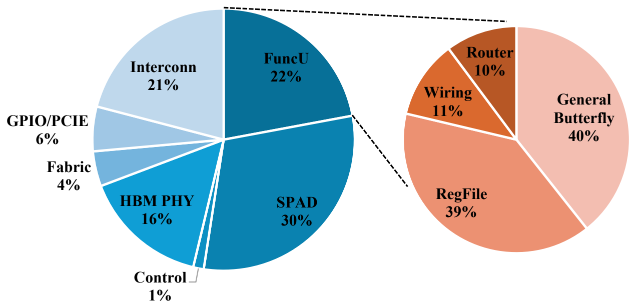


Fig. 9. The area breakdown of UFC.

naturally supporting most FHE operations. Specifically, polynomials in the evaluation form need element-wise operations so that the interleaved layout enables each PE to process the local data independently. Furthermore, polynomials in the original form only require unary functions (e.g., decomposition, base conversion, etc.) that can be processed in the continuous layout.

B. Parallel Scheduling

The small polynomial packing requires a sufficient number of parallel operations on independent data in the logic-scheme FHE to achieve full hardware utilization. Previous work [41] recognizes various parallelisms that can be exploited, including 1) test-vector parallelism (TvLP, different bootstrapping operations), 2) polynomial-level parallel (PLP, polynomials in one RLWE ciphertext), 3) coefficient-level parallelism (CLP, coefficients in each polynomial), and 4) column-level parallelism (CoLP, decomposed polynomials in the external product). In UFC, the wide lane design supports high-degree coefficient-level parallelism, and all other three parallelisms can exploit the small-polynomial packing, as shown in Figure 8.

Similar to previous work [41], UFC prioritizes these three parallelisms in the order of 1) TvLP, 2) PLP, and 3) CoLP, but with different considerations. First, due to the almost identical operation patterns of functional bootstrapping on different test vectors, TvLP can effectively reuse the bootstrapping key across different ciphertexts, resulting in the lowest memory bandwidth stress. However, TvLP depends on the application. When TvLP is not sufficient, we prioritize PLP over CoLP because CoLP requires a shuffling process to place the decomposed polynomials in the continuous layout (Section V-C). On the other hand, PLP only needs static allocation of polynomials in RLWE ciphertext.

C. Memory Allocation and Data Shuffling

Due to the parallel scheduling with small polynomial packing, static memory allocation may not be sufficient to load and store data correctly between on-chip and off-chip memory, where each HBM channel connects to a specific scratchpad partition in UFC. Therefore, the near-data LWEU contains a crossbar network and a dispatch unit, as introduced in Section IV-B4, supporting arbitrary data shuffling between HBM channels and on-chip scratchpad.

VI. Experimental Setup

A. Hardware Design and Modeling

The hardware components for UFC shown in Figure 9, are synthesized, optimized for an advanced commercial technology. To comply with disclosure restrictions, we scale the power and area to 7nm for comparison with prior works [26]. We calculate the scaling factor based on previous work [47] and the differences between our technology and previously-used open-sourced technology [13], [46]. We adopt an optimized Montgomery multiplier design for moduli $q_i = -1 \; mod \; 2^{16}$, similar to F1 [43]. Our analysis is projected to 1GHz. Table II shows the detailed configuration of UFC's hardware components, while Figure 9 has the area breakdown. All functional units use 32-bit data and double-scaling operations for arbitrary FHE parameters [26]. Processing elements mainly consist of butterfly

TABLE III
FHE PARAMETER SETTINGS

CKKS Parameters	
C1	$N = 2^{16}, dnum = 3, logPQ = 1555$
C2	$N = 2^{16}, dnum = 3, logPQ = 1764$
C3	$N = 2^{16}, dnum = 4, logPQ = 1679$
TFHE Parameters	
T1	$n = 500, N = 2^{10}, g_k = 2$
T2	$n = 630, N = 2^{10}, g_k = 3$
T3	$n = 592, N = 2^{11}, g_k = 3$
T4	$n = 991, N = 2^{14}, g_k = 2$

units, register files, and circuits for the on-chip interconnect. Due to the simplicity of the functional units, which is key to achieving high hardware utilization for different schemes, interconnect takes up a significant part of the chip. Overall, UFC has a similar chip area and power consumption to the state-of-the-art accelerator of SIMD-schemes [26], with better performance on SIMD-scheme workloads and the capability of supporting logic-scheme workloads.

B. Software Implementation

We use the OpenFHE library, which supports CKKS, TFHE, and scheme switching for hybrid FHE programs [2]. We develop a tracing tool that can generate traces of various FHE operations at the granularity of ciphertext for different applications. We input the generated traces to a Python-based compiler to generate the hardware instructions with compiler-level optimizations.

C. Simulation Infrastructure

We develop a cycle-accurate simulation framework, which takes FHE hardware instructions as the input, to evaluate UFC based on data from synthesis. Furthermore, we exploit the simulation framework to evaluate our baseline accelerators [26], [41]. Specifically, we implement separate performance models for different operation macros supported by the pipelined hardware in previous works [26], [41]. We use the architectural parameters reported in previous papers to develop the performance model based on the upper-bound throughput (e.g., 1,024 elements per cycle NTT in SHARP [26]). We reuse our scratchpad management policy to handle all data replacements in a word-size ciphertext granularity (e.g., one 36-bit RNS limb for SHARP [26]). The unified simulation framework makes a fair comparison because all architectures use the same instruction traces that depend on the FHE algorithm implementation, compiler-level instruction generation, memory operation scheduling, etc. We note that our traces exhibit higher HBM utilization (Section VII-B) than that reported in SHARP [26]. To avoid the impact of under-optimized memory operation scheduling when evaluating SHARP [26], our SHARP simulation assumes a 288 MB (36-bit data) scratchpad to achieve similar function unit utilization as the reported values [26].

D. Workloads and FHE Parameters

We test various workloads using CKKS and TFHE:

1) SIMD-scheme Workloads: We evaluate four CKKS workloads at 128-bit security, which are used by prior accelerators [26], [27], [44], as shown in Table III.

Logistic Regression (HELR) [20]: This workload has 30 iterations of homomorphic logistic regression, where each iteration trains 1024 samples with 256 features as a batch. The multiplication depth is deep, requiring several bootstrapping operations.

ResNet-20 [29]: The ResNet-20 is a homomorphic neural network inference for one CIFAR-10 image classification. The network is deep with multi-channel convolutions, matrix multiplications, and approximated ReLU function.

Sorting [22]: Sorting uses 2-way bitonic sorting on an array with 16,384 elements, the same as that used in SHARP [26].

Bootstrapping [21]: We evaluate the bootstrapping algorithm using a similar framework as previous work [21], [27], which requires 30 (32-bit) levels of bootstrapping. We adopt the minimum-key method used in previous work [27] to reduce the rotation keys used in bootstrapping.

2) Logic-scheme Workloads: We evaluate several TFHE workloads using four different parameter sets (Table III) as the state-of-the-art TFHE accelerator [41], including functional bootstrapping throughput tests and ZAMA-NNs [12].

3) Hybrid Workload: The hybrid workloads include CKKS and TFHE operations with scheme-switching process [2], [33]. We implement a k-nearest neighbor search [14] as the end-to-end hybrid benchmark for evaluation. To compare the efficiency of the hybrid workloads, we assume the baseline system has one SHARP [26] and one Strix [41] simultaneously and uses the 16 PCIe5 lanes to handle data communication between these different chips. We scale the delay, power, and area of Strix [41] to 7 nm for a fair comparison using the scaling methods [47].

VII. EVALUATION

In this section, we first compare UFC to the state-of-the-art FHE accelerators for CKKS [26] and TFHE [41] in performance, power, and area. Then, we show the results of UFC's comprehensive design space exploration to provide insights on trade-offs in various architectural configurations. Furthermore, we evaluate the sensitivity of our design to various optimization parameters presented in previous sections.

A. Comparison to State of the Art Accelerators

1) SIMD-scheme Workloads: Figure 10(a) shows the performance of various workloads on CKKS, as compared to SHARP [26]. Overall, UFC outperforms SHARP by 1.1× and 1.4× in terms of delay and energy, resulting in a 1.5× improvement on the energy-delay product (EDP). Considering the chip area, UFC achieves 1.6× better energy-delay-area product (EDAP) than SHARP. We discuss more details about such comparison in Section VII-B and VII-C.

2) Logic-scheme Workloads: Figure 10(b) shows the performance of two workloads on TFHE, as compared to Strix [41]. Based on our evaluation, UFC is 6× faster than Strix on TFHE workloads because of its higher throughput,

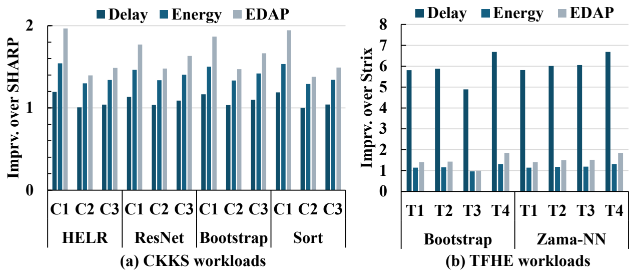


Fig. 10. The comparison to prior scheme-specific accelerators.

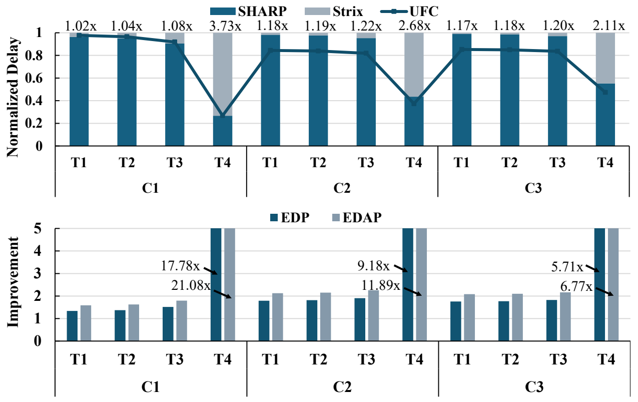


Fig. 11. The comparison on hybrid FHE program for k-NN.

thanks to the large chip area designed for large algorithmic parameters. Specifically, Strix has 8 clusters, each adopting a fully pipelined 14-stage NTT with 4 copies. Therefore, the total butterfly units in Strix is 1792, which is 4.6× less than that in UFC. In terms of efficiency, UFC consumes 1.2× less energy and yields 1.5× better EDAP than Strix [41]. Section VII-D discusses the impact of different TFHE implementations used by UFC and Strix [41]

3) Hybrid Workloads: Figure 11 shows the performance of UFC and the composed SHARP+Strix system on k-NN with hybrid FHE schemes [14]. In the small TFHE parameter settings (T1 - T3), CKKS operations take a significant portion of time, where UFC provides 1.04× speedup over the composed system. The speedup becomes more evident on large TFHE parameters (i.e., T4), which is 2.8× on average, because UFC significantly outperforms Strix. We also compare the energy efficiency (EDP) and energy-area efficiency (EDAP) of UFC and the composed baseline system. We calculate the area and power of the composed system by scaling the Strix to 7 nm. Based on our results, UFC is 3.1× and 3.7× more efficient than the baseline system in terms of EDP and EDAP.

B. Utilization Analysis

Figure 12 shows the hardware utilization breakdown in UFC on different workloads. We observe that the utilization of processing elements, NoC, and HBM are 65%, 20%, and 69% in CKKS workloads. The high PE utilization results

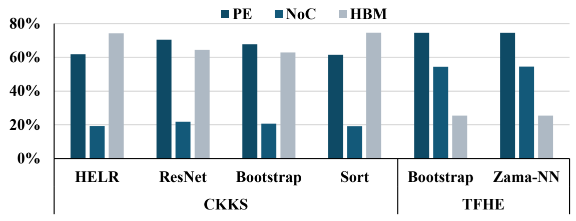


Fig. 12. Utilization of key UFC components.

TABLE IV
ARCHITECTURE COMPARISON BETWEEN SHARP [26] AND UFC.

	SHARP [26]	UFC
Word length	36-bit	32-bit
Core frequency	1GHz	1GHz
# of lanes	1,024	16,384
Off-chip memory BW	1TB/s	1TB/s
On-chip memory Cap	180+18MB	256MB+18MB
On-chip memory BW	36TB/s + 36TB/s	64TB/s + 64TB/s
Global NoC BW	1,024 w/c	32,768 w/c
NTTU throughput	1,024 w/c	1,024 w/c
NTTU bisection BW	128 w/c	32,768 w/c
BConv throughput	16,384 w/c	16,384 w/c
ELEW throughput	2,048 w/c	16,384 w/c

from its versatility for various operations during the execution, including element-wise operations, base conversion, and NTT. Furthermore, UFC achieves higher PE and NoC utilization (75% and 55%, respectively) on TFHE workloads than CKKS workloads because the 256MB on-chip scratchpad is sufficiently large for small ciphertexts in TFHE workload (an average 25% utilization in HBM).

C. Comparison to SoTA CKKS Accelerator

Table IV shows the detailed comparison between SHARP [26] and UFC. Compared to SHARP, UFC supports similar throughput for NTTs and base conversions and higher throughput for element-wise operations. We note that UFC adopts a versatile PE design that can support 16,384 and 8,192 computations for BConv/ELEW and NTT, respectively. On the contrary, SHARP has separate function units for different operations. UFC achieves a similar overall utilization of multipliers as compared to SHARP. Furthermore, UFC utilizes the register files with high on-chip memory bandwidth for data communications across different operations. Such on-chip memory bandwidth is higher than SHARP, which exploits all-to-all NoC to transform data between evaluation and coefficient forms.

D. FFT vs. NTT in TFHE Applications

The key difference between UFC and the baseline TFHE accelerator [41] is the choice of the ciphertext modulus, where UFC supports NTT-friendly primes and Strix [41] supports powers of two, both 32-bit integer. This difference leads to different function unit designs. Specifically, UFC, similar to prior CKKS accelerators [26], [27], [43], [44],

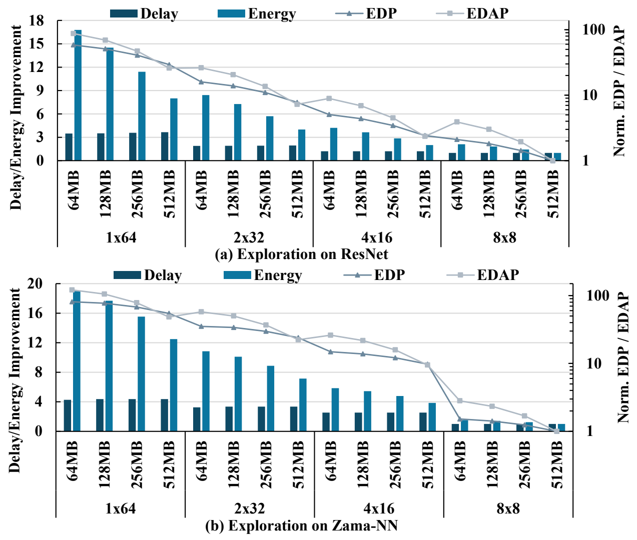


Fig. 13. The design space exploration on CG-NTT configurations.

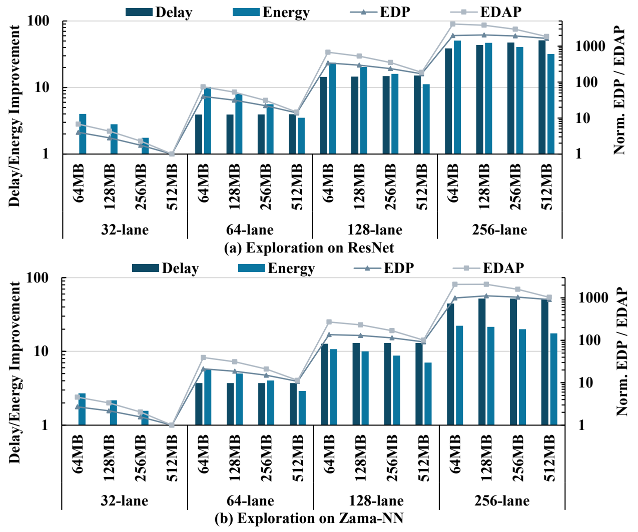


Fig. 14. The design space exploration on throughput.

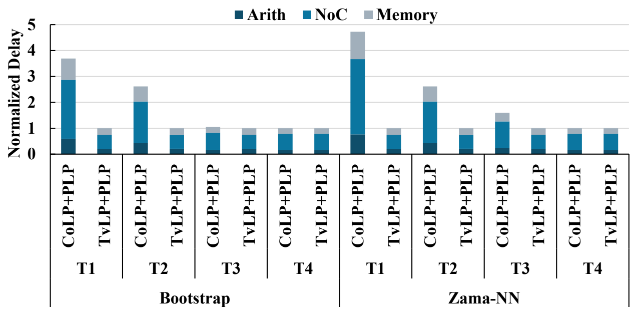


Fig. 15. Performance gain of small-polynomial packing with CoLP and TvLP.

adopts modular multiplier as the basic arithmetic unit and uses NTT for polynomial multiplication. Strix [41] consists of normal 32-bit arithmetic units with 64-bit FFT units due to the double-precision requirement for FFT. Compared to FFT, NTT provides accurate results but requires extra modular reduction [25]. We note that NTT and FFT implementations are both standard in open-sourced FHE libraries [1], [2], [12], which support the same application-level functionality.

E. Design Space Exploration

We conduct a design space exploration to determine the architectural parameters of UFC.

1) Constant-Geometry NTT Hardware: There exists a design space to apply different sizes of CG-NTT network in UFC. For example, instead of connecting the 64 PEs in one CG-NTT network, we can connect each 32 PEs with a separate CG-NTT network (2x smaller CG-NTT network). Therefore, we explore the number of separate CG-NTT networks in Figure 13. For each configuration, we also explore the scratchpad capacity. As shown in the Figure, a single large CG-NTT network constantly outperforms systems with more CG-NTT networks. We also observed that UFC with a smaller scratchpad provides better EDP and EDAP. We choose 256MB for comparison to maximize the UFC performance with a similar chip area to the SHARP. However, the high efficiency of UFC with a smaller scratchpad is a promising alternative if the overall cost becomes the major concern.

2) Throughput: We explore the throughput of UFC by varying the number of lanes in each PE. Increasing the number of lanes in each PE by 2× leads to 2× higher throughput for both arithmetic computation and NTT data shuffling, with at least 2× larger hardware area for ALUs and CG-NTT network. Figure 14 shows the results of design space exploration by exploring the number of lanes and the scratchpad capacity. The results show that UFC achieves better EDP and EDAP on configurations with more lanes on the hardware, showing good scalability of the proposed architecture.

F. Compiler-level Optimization

As introduced in Section V-A, UFC can exploit various levels of parallelism for small polynomials (in the logic-based schemes). Figure 15 shows the performance comparison between CoLP and TvLP, both working with PLP in UFC. Based on our evaluation, TvLP significantly outperforms CoLP in the small parameter settings. This is because 1) CoLP does not have sufficient independent data to fully utilize the hardware, and 2) CoLP requires more bootstrapping keys on the chip, significantly increasing the memory overhead. However, the benefits of TvLP shrink when the parameter size is increased because fewer parallel polynomials can be packed, and TvLP requires a larger on-chip working set to accumulate multiple partial products.

VIII. Related Works

A. SIMD-scheme FHE Accelerators

There has been a significant body of research proposing customized accelerator SIMD FHE schemes, most of which target the CKKS scheme [26]–[28], [42]–[44], the same as the target SIMD scheme used in this work. Specifically, HEAX [42] utilized low-throughput function units with deeply customized hardware pipelines to support FHE applications. F1 [43] utilizes a novel 4-stage pipeline design for NTT operations, providing high throughput with less hardware specialization. CraterLake [44] further optimizes the design of F1 [43] by using a fixed permutation network to shuffle data in function units that are much wider than those in F1. Furthermore, CraterLake [44] adds more hardware specialization to accommodate the operations (e.g., key-switching) in the latest CKKS algorithms. Different from F1 and Craterlake, ARK [27] and SHARP [26] adopt a different architecture template where the coefficient-form and evaluation-form function units are integrated together with all-to-all interconnect to handle data dependency between two forms. Furthermore, both ARK [27] and SHARP [26] apply significant algorithm-hardware co-optimizations to select the most efficient parameters for both hardware architecture and algorithm implementation. However, both ARK [27] and SHARP [26] share several similar design philosophies to F1 [43] and CraterLake [44], including the multi-stage pipelined NTT unit, customized logic for base conversion, and deeply pipelined function units for high throughput with limited throughput.

UFC differs from all these previous works in the fundamental architecture design philosophy. UFC minimizes the hardware specialization for a single scheme (e.g., CKKS) by only adopting general function units for wide data, which is several times wider than prior CKKS accelerators. UFC exploits several algorithm-hardware co-design techniques to tackle the challenge of data shuffling in wide FHE function units, achieving better performance and cost-efficiency than prior architecture. There is one previous FHE accelerator, BTS [28], that shares some similarities in the architecture design with UFC. BTS [28] organizes 2048 simple PE, each with 1 butterfly unit and arithmetic unit, in a 32×64 2D array, and exploits crossbar interconnects in the horizontal and vertical direction to handle cross-PE dependency. However, BTS's design cannot scale to the throughput of UFC because of the expensive interconnection networks. Furthermore, UFC supports high-throughput processing for logic FHE schemes, which are not supported by all these prior accelerators.

B. Logic-scheme FHE Accelerators

There have been multiple accelerators for TFHE, including TVE [18], MATHCA [23], FPT [48], and Strix [41]. All these accelerators adopt specialized hardware pipelines for key operations in TFHE. For example, MATCHA [23] and FPT [48] balance the throughput of external products by adopting different numbers of NTT and iNTT units. Both Strix [41] and FPT [48] adopt streaming architecture to connect different function units for high-throughput TFHE operations. However, all these accelerators only target TFHE with hardware specialization, lacking the support for SIMD FHE schemes. Furthermore, their designs target small parameter settings in TFHE, so the scalability of the hardware architecture is limited to support larger data structures required by SIMD FHE schemes. On the contrary, UFC can support various FHE schemes and provide better performance and cost-efficiency due to the algorithm-hardware co-designs.

C. Accelerators for Lattice-based Cryptography

In addition to FHE-specific accelerators, previous works propose several accelerators for lattice-based cryptography [3], [15], [30], [38], [39], [49]. These accelerators exploit different technologies to accelerate fundamental operations in lattice-based cryptography algorithms, such as NTT and modular arithmetic. These accelerators propose several cost-efficient designs for NTTs. For example, CoHA-NTT [15] and Sapphire [3] exploit the constant-geometry NTT algorithm to reduce the hardware cost. MeNTT [30], BP-NTT [49], and NTT-PIM [39] utilize processing in-memory (PIM) technologies to support high-throughput NTT operations. However, these accelerators lack consideration in both hardware and software for the end-to-end FHE applications.

IX. Conclusion

This work proposes a novel accelerator, UFC, with a unified architecture that supports various FHE schemes. UFC consists of general function units that support operations in different FHE schemes. To improve the scalability of the proposed accelerator, we exploit several algorithm-hardware co-designs to minimize the hardware cost of increasing the number of general function units. UFC also adopts compiler-level optimizations to achieve high hardware utilization on various FHE parameter settings. Our evaluation of workloads in various FHE schemes shows that UFC provides 1.1-6.0× speedup and 1.5-1.6× delay-energy-area efficiency improvement over state-of-the-art scheme-specific FHE accelerators.

Acknowledgment

This work was supported by PRISM and CoCoSys, centers in JUMP 2.0, an SRC program sponsored by the Defense Advanced Research Projects Agency. This work was also supported by National Science Foundation (NSF) funding #2112167, #2112665, #2003279, #2120019, #2211386, #2052809, #1911095. This research was, in part, funded by the Defense Advanced Research Projects Agency through contract HR0011-21-3-0003. The views, opinions, and findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] "PALISADE Lattice Cryptography Library (release 1.11.5)," https://palisade-crypto.org/, 2021.

[2] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915. [Online]. Available: https://eprint.iacr.org/2022/915

[3] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *arXiv preprint arXiv:1910.07557*, 2019.
[4] S. Bian, Z. Zhao, Z. Zhang, R. Mao, K. Suenaga, Y. Jin, Z. Guan, and J. Liu, "Heir: A unified representation for cross-scheme compilation of fully homomorphic computation," Cryptology ePrint Archive, Paper 2023/1445, 2023, https://eprint.iacr.org/2023/1445. [Online]. Available: https://eprint.iacr.org/2023/1445
[5] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Chimera: Combining ring-lwe-based fully homomorphic encryption schemes," *Journal of Mathematical Cryptology*, vol. 14, no. 1, pp. 316–338, 2020.
[6] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.
[7] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.
[8] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 360–384.
[9] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.
[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.
[11] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, vol. 15, 2020.
[12] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," Cryptology ePrint Archive, Paper 2021/091, 2021, https://eprint.iacr.org/2021/091. [Online]. Available: https://eprint.iacr.org/2021/091
[13] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.
[14] K. Cong, R. Geelen, J. Kang, and J. Park, "Revisiting oblivious top-k selection with applications to secure k-nn classification," Cryptology ePrint Archive, Paper 2023/852, 2023, https://eprint.iacr.org/2023/852. [Online]. Available: https://eprint.iacr.org/2023/852
[15] K. Derya, A. C. Mert, E. Öztürk, and E. Savaş, "Coha-ntt: A configurable hardware accelerator for ntt-based polynomial multiplication," *Microprocessors and Microsystems*, vol. 89, p. 104451, 2022.
[16] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.
[17] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.
[18] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, "An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus," in *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*, 2021.
[19] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.
[20] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 9466–9471.
[21] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.
[22] S. Hong, S. Kim, J. Choi, Y. Lee, and J. H. Cheon, "Efficient sorting of homomorphic encrypted data with k-way sorting network," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4389–4404, 2021.
[23] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 235–240. [Online]. Available: https://doi.org/10.1145/3489517.3530435
[24] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1209–1222.
[25] M. Joye and M. Walter, "Liberating tfhe: programmable bootstrapping with general quotient polynomials," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 1–11.
[26] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589053
[27] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254.
[28] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," *arXiv preprint arXiv:2112.15479*, 2021.
[29] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.
[30] D. Li, A. Pakala, and K. Yang, "Mentt: A compact and efficient processing-in-memory number theoretic transform (ntt) accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 5, pp. 579–588, 2022.
[31] Z. Liang and Y. Zhao, "Number theoretic transform and its applications in lattice-based cryptosystems: A survey," *arXiv preprint arXiv:2211.13546*, 2022.
[32] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *International Conference on Cryptology and Network Security*. Springer, 2016, pp. 124–139.
[33] W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1057–1073.
[34] V. Lyubashevsky, C. Peikert, and O. Regev, "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, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 2010, pp. 1–23.
[35] K. Matsuoka, Y. Hoshizuki, T. Sato, and S. Bian, "Towards better standard cell library: Optimizing compound logic gates for tfhe," in *Proceedings of the 9th on Workshop on Encrypted Computing; Applied Homomorphic Cryptography*, ser. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 63–68. [Online]. Available: https://doi.org/10.1145/3474366.3486927
[36] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 86, 2020.
[37] G. Miel, "Constant geometry fast fourier transforms on array processors," *IEEE transactions on computers*, vol. 42, no. 3, pp. 371–375, 1993.
[38] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "Cryptopim: In-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
[39] J. Park, S. Lee, and J. Lee, "Ntt-pim: Row-centric architecture and mapping for efficient number-theoretic transform on pim," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.
[40] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 252–264, 1968.
[41] A. Putra, Prasetiyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1319–1331. [Online]. Available: https://doi.org/10.1145/3613424.3614264

[42] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, “Heax: An architecture for computing on encrypted data,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[43] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, “F1: A fast and programmable accelerator for fully homomorphic encryption,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252. [Online]. Available: https://doi.org/10.1145/3466752.3480070

[44] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, “Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 173–187. [Online]. Available: https://doi.org/10.1145/3470496.3527393

[45] “Microsoft SEAL (release 3.7),” https://github.com/Microsoft/SEAL, Sep. 2021, microsoft Research, Redmond, WA.

[46] A. Shafaei, Y. Wang, X. Lin, and M. Pedram, “Fincacti: Architectural analysis and modeling of caches with deeply-scaled finfet devices,” in *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2014, pp. 290–295.

[47] A. Stillmaker, Z. Xiao, and B. Baas, “Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm,” *VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4*, vol. 4, p. m8, 2011.

[48] M. Van Beirendonck, J.-P. D’Anvers, F. Turan, and I. Verbauwhede, “Fpt: A fixed-point accelerator for torus fully homomorphic encryption,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 741–755. [Online]. Available: https://doi.org/10.1145/3576915.3623159

[49] J. Zhang, M. Imani, and E. Sadredini, “Bp-ntt: Fast and compact in-sram number theoretic transform with bit-parallel modular multiplication,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.