Journal of Computer Science and Technology

Indexed in: SCIE, EI, Scopus, INSPEC, DBLP,CSCD, etc.
Sponsored by: ICT, CAS & CCF

JCST Papers

Only for Academic and Non-Commercial Use

Thanks for Reading!

JCST WeChat Subscription Account

JCST URL: https://jcst.ict.ac.cn

SPRINGER URL: https://www.springer.com/journal/11390

E-mail: jcst@ict.ac.cn

Online Submission: https://mc03.manuscriptcentral.com/jcst

Twitter: JCST_Journal

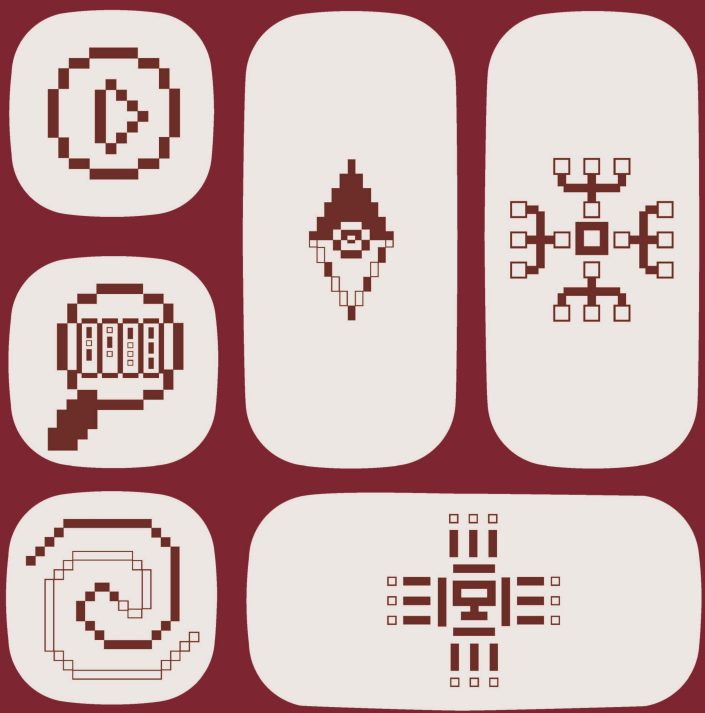
LinkedIn: Journal of Computer Science and Technology

计算机科学技术学报（英文）
PRINT ISSN: 1000-9000
ONLINE ISSN:1860-4749
CODEN JCTEEM
Vol. 41 No.3 May 2026

Journal of Computer Science and Technology

Perspective: A Processor Approach to Fast Fully Homomorphic Encryption
Zhi-Wei Xu

Cover Article: Homomorphic Processing Unit
Ying-Hao Yang, Xi-Cheng Xu, Hang Lu, and Xiao-Wei Li



Yang YH, Xu XC, Lu H *et al.* Homomorphic processing unit. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, 41(3): 845-861, May 2026. DOI: 10.1007/s11390-026-5125-0, CSTR: 32374.14.s11390-026-5125-0

Homomorphic Processing Unit

Ying-Hao Yang (杨英豪), *Member, CCF*, Xi-Cheng Xu (徐熙程), Hang Lu* (路　航), *Member, CCF, IEEE* and Xiao-Wei Li (李晓维), *Fellow, CCF, Senior Member, IEEE*

State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

E-mail: yangyinghao@ict.ac.cn; xuxicheng20@mails.ucas.ac.cn; luhang@ict.ac.cn; lxw@ict.ac.cn

Received March 6, 2025; accepted June 3, 2026.

Abstract Fully homomorphic encryption (FHE) enables computation over encrypted data without decryption, ensuring data confidentiality throughout the processing pipeline. However, the complexity and heterogeneity of FHE schemes like CKKS, BFV, and TFHE pose challenges for unified hardware design. In this paper, we propose the Homomorphic Processing Unit (HPU), a general-purpose platform supporting multiple FHE schemes. Unlike fixed-function accelerators, HPU is implemented as a RISC-V extension and introduces a dedicated homomorphic instruction set architecture (H-ISA), comprising micro-instructions for minimal compute kernels and macro-instructions for integration with general-purpose models. Core operations like the number-theoretic transform (NTT) and automorphism are abstracted into a unified instruction layer, mapped onto specialized homomorphic compute units. HPU integrates a collaborative computation and storage design, utilizing a polynomial-level instruction set and a polynomial-granular memory management unit (MMU) to optimize memory and reduce data movement. By combining a complete ISA with a general-purpose computing core, HPU can achieve efficient computing, control, and scheduling. We evaluate the HPU prototype using four multi-FHE benchmarks on field-programmable gate array (FPGA) and 7 nm ASIC. The key results are as follows. 1) HPU achieves up to 1596x and 1174x speedup over CPU for CKKS-like and TFHE operations, respectively. 2) Compared with SOTA FPGA solutions for CKKS and TFHE, HPU achieves 1.36x and 1.2x improvement, respectively. 3) HPU's ASIC implementation achieves 1.36x speedup over state-of-the-art TFHE accelerators for long short-term memory (LSTM).

Keywords fully homomorphic encryption (FHE), Homomorphic Processing Unit (HPU), privacy computing

1 Introduction

Fully homomorphic encryption (FHE) is widely regarded as one of the most promising privacy-preserving technologies. Using FHE, sensitive data, such as financial records, medical histories, personal behavioral data, and inputs for secure deep learning inference, can be encrypted by the data owner and transmitted to a third-party service provider for direct computation on the encrypted data[1–3]. Throughout the computation process, the data remain inaccessible to the service provider in plaintext form, thereby mitigating the risk of privacy leakage.

FHE has undergone rapid development over the past decade. In 2009, Gentry proposed the first practical FHE scheme, which supports an arbitrary number of homomorphic additions and multiplications. Subsequently, several influential FHE schemes were developed to improve FHE's efficiency and functionality, including BFV[4], BGV[5], CKKS[6], and TFHE[7]. BGV, BFV, and CKKS are designed for arithmetic circuits and support single instruction multiple data (SIMD) computation, whereas TFHE is based on Boolean circuits and operates on individual data elements. These schemes also differ in their computation models. BGV and BFV provide exact computation, producing decrypted results identical to those obtained from plaintext computation, whereas CKKS

Cover Article

The work was supported in part by the Chinese Academy of Sciences under Grant No. XDB0660102 and the National Natural Science Foundation of China under Grant No. 62172387.

*Corresponding Author

supports approximate computation, allowing controlled precision loss in exchange for improved efficiency. These FHE schemes have been widely adopted across various applications. CKKS, BFV, and BGV are commonly used in machine learning, deep learning, private set intersection, and private information retrieval[2, 8, 9]. In contrast, TFHE supports arbitrary data mappings through lookup tables (LUTs) and programmable bootstrapping (PBS), which simultaneously refreshes ciphertext noise. Consequently, it is particularly well suited for applications with small plaintext spaces and frequent evaluations of nonlinear functions[10–12]. Moreover, combining multiple FHE schemes to exploit their complementary strengths has emerged as an effective approach for improving computational efficiency. For example, PEGASUS[11] leverages the SIMD capability of CKKS and the PBS functionality of TFHE to efficiently support both linear and nonlinear computations across diverse applications. Similarly, AFBoot[12], which combines BFV and TFHE, employs LUTs to achieve batch noise refreshing for learning-with-errors (LWE) ciphertexts. Despite these advances, a substantial gap remains between theoretical research and practical deployment. Even when using highly optimized FHE libraries, FHE applications typically execute 10 000x–100 000x slower than their unencrypted counterparts on CPUs.

Consequently, specialized hardware accelerators are required to improve the performance of FHE schemes. In addition, applications that integrate multiple FHE schemes demand accelerator architectures capable of supporting diverse computation models. However, most existing studies on FHE acceleration focus on optimizing individual schemes in isolation. Trinity[13] and UFC[14] are representative multi-scheme accelerators designed to support both CKKS and TFHE. Trinity allocates dedicated hardware resources to each scheme, whereas UFC modifies the FHE computation flow to utilize a unified processing-element (PE) array, albeit at the cost of increased algorithmic complexity. Despite these advances, such approaches focusing on FHE acceleration achieve only partial unification, as they fail to fully exploit the underlying commonalities among FHE algorithms. These accelerator architectures are fundamentally designed as auxiliary computing devices and therefore rely on tight coordination with a host CPU to execute FHE workloads. Consequently, both the accelerator microarchitecture and the host-side software stack, including compilers, programming frameworks, drivers, and runtime systems, require extensive customization and co-design. Under this paradigm, switching between or integrating multiple FHE schemes becomes difficult to maintain and scale because of system fragmentation, often resulting in significant performance degradation. Furthermore, the separation of control and computation limits compiler optimization opportunities. Frequent interactions and synchronizations between the host and the accelerator are required, making unified optimization of control flow, computation, and scheduling challenging. Given the increasing diversity of FHE schemes and applications, a promising direction is the development of a general-purpose homomorphic processor based on a unified instruction set architecture (ISA). By abstracting common computational kernels, defining a hierarchical ISA, and co-designing the architecture with a general-purpose control unit, such a processor can efficiently support multiple FHE schemes while providing a unified programming interface that abstracts low-level computational details. This approach can reduce programming and compilation complexity, improve software portability, and enhance the usability and accessibility of FHE systems.

In this work, we propose the Homomorphic Processing Unit (HPU), which adopts a fundamentally different architectural philosophy from conventional FHE accelerators. HPU defines a hierarchical homomorphism-oriented ISA (H-ISA) that abstracts scheme-specific computational differences and provides a unified interface for programming frameworks. By tightly integrating H-ISA with the control capabilities of a general-purpose compute core, HPU establishes an FHE-oriented processor model that enables joint optimization of computation, control, and scheduling. The main contributions of this paper are summarized as follows.

1) We propose HPU, a processor-style multi-scheme FHE platform implemented as a RISC-V extension with a hierarchical H-ISA. Unlike conventional decoupled accelerator architectures, HPU integrates a complete ISA with a general-purpose compute core, elevating FHE execution from fixed-function offloading to a programmable processor model. The hierarchical H-ISA provides a unified instruction-level abstraction across heterogeneous FHE schemes, enabling efficient execution of both multi-scheme and hybrid workloads on a single hardware platform.

2) We design a two-level H-ISA that separates a

minimal set of homomorphic computational kernels from a general-purpose programming interface, enabling instruction-level reuse and pipelined execution. Core homomorphic operations are abstracted as reusable micro-instructions and mapped to specialized compute units, while macro-instructions integrate these operations into a general-purpose programming model. This separation facilitates fine-grained task decomposition, cross-scheme reuse, and pipeline-friendly execution without requiring scheme-specific control flows to be hardwired into the hardware.

3) We introduce an HPU microarchitectural paradigm that enables the joint optimization of computation, control, and scheduling within a unified processor architecture. By tightly integrating H-ISA with a general-purpose compute core, HPU performs instruction-level execution and scheduling autonomously, eliminating the need for host-side runtime orchestration. This design mitigates system fragmentation in multi-scheme environments and expands the architectural optimization space under constrained on-chip resources.

4) We implement and evaluate HPU prototypes on both FPGA and 7 nm ASIC platforms, demonstrating strong performance across multiple FHE benchmarks. Across four domain-specific benchmarks, the FPGA prototype achieves speedups of up to 1 596 x and 1 174x over CPU implementations for CKKS-like and TFHE operations, respectively, while delivering improvements of up to 1.36x and 1.2x over state-of-the-art (SOTA) FPGA accelerators for CKKS and TFHE, respectively. The ASIC implementation further achieves a 1.36x speedup over a SOTA TFHE accelerator on the LSTM workload, demonstrating that an ISA-centric, processor-style architecture can simultaneously provide generality and high performance.

2 Background and Motivation

2.1 FHE Basic Operations

To unify computational patterns across different FHE schemes and abstract scheme-specific differences through instruction definitions, we first analyze the primitive operations underlying FHE. This analysis enables the identification and extraction of a homomorphic computational minimum kernel (CMK) that captures the common computational characteristics of different FHE schemes, thereby providing the foundation for a complete H-ISA definition. Since the CMK of CKKS-like schemes has already been abstracted in Poseidon[15], this section focuses on the analysis of TFHE. The most critical operation in TFHE is PBS, which simultaneously evaluates an arbitrary function through a lookup table while refreshing ciphertext noise[7]. PBS consists of five fundamental TFHE operations: modulus switching (MS), external product (EProduct), blind rotation (BRotation), sample extraction (SE), and key switching (Keyswitch).

1) *MS.* The first step, modulus switching, rescales the coefficients of an LWE ciphertext by a factor of $2N/q$ and rounds the results to switch the modulus from q to $2N$. Formally, this operation is expressed as $\tilde{a}_i = \lfloor 2Na_i/q \rceil$ and $\tilde{b} = \lfloor 2Nb/q \rceil$. `MS` is a lightweight arithmetic kernel that performs power-of-two scaling followed by quotient-related reduction and rounding. Since N is a power of two, multiplication by $2N$ can be implemented as a shift operation, whereas division by q and rounding are abstracted into a shared quotient-and-reduction primitive. We denote this CMK primitive as SQM (Shared Quotient Modulo), which can be reused across different FHE schemes whenever quotient computation and modular reduction are required.

2) *EProduct.* In TFHE, `EProduct` performs the multiplication between the accumulator ciphertext (ACC) and the bootstrapping key (BSK). The ACC is an RLWE (ring learning with errors) ciphertext, whereas the BSK is an RGSW ciphertext encrypted under the secret key associated with the input LWE ciphertext. Consequently, the BSK consists of n RGSW ciphertexts, i.e., BSK_1, BSK_2, ..., BSK_n ($\in \mathbb{R}_{(N,\,p)}[X]^{2l}$), where l denotes the decomposition level. To control noise growth during ciphertext multiplication, each polynomial is decomposed as $Decomp_{(\alpha,\,l)}(M(x)) = (M_1(x), \ldots, M_l(x))$, such that $M(x) = M_1(x)(q/\alpha^1)+M_2(x)(q/\alpha^2)+\ldots+M_l(x)(q/\alpha^l)$, where $M_i(x) = M(x)/(q/\alpha^i)$. At a high level, this decomposition process involves repeated scaling, division, and rounding operations, which are conceptually similar to those used in `MS`. `EProduct` can be decomposed into a small set of reusable primitives: 1) modular arithmetic and polynomial-domain multiplication primitives, i.e., modular addition (MA), modular multiplication (MM), and number theoretic transform (NTT); 2) coefficient-wise extraction and rotation operations represented by the sample extraction (SE); and 3) division-and-rounding-based scaling op-

erations captured by the shared quotient-and-modulo (SQM), consistent with MS.

3) *BRotation.* Blind rotation (BRotation) is implemented through a sequence of EProduct operations, each corresponding to one term in the decryption expression of the input LWE ciphertext, i.e., $\tilde{a}_i s_i$. Consequently, n EProducts compute the accumulated value $\tilde{a}_1 s_1 + \tilde{a}_2 s_2 + \ldots + \tilde{a}_n s_n$. A final ciphertext rotation by $\tilde{b}$ is then applied to complete the blind rotation, yielding $\tilde{a}_1 s_1 + \tilde{a}_2 s_2 + \ldots + \tilde{a}_n s_n + \tilde{b}$. BRotation is a higher-level operation that repeatedly invokes the EProduct compute kernel, followed by a final rotation and offset update. BRotation introduces no additional CMK primitives beyond those required by EProduct. Therefore, BRotation can be represented using the same set of CMK building blocks: MA, MM, NTT, inverse NTT (INTT), SE, and SQM.

4) *SE.* Following blind rotation, the refreshed message is embedded in the first coefficient of the RLWE ciphertext polynomial, requiring SE to isolate the message. SE can extract up to N LWE ciphertexts from a single RLWE ciphertext, where N denotes the polynomial degree. Although only one extracted ciphertext is typically used in standard TFHE bootstrapping, multi-scheme applications (e.g., CKKS-TFHE) require interoperability between RLWE- and LWE-based ciphertext representations and may therefore involve the extraction of multiple ciphertexts. Accordingly, we define SE as a homomorphic instruction that supports full-scale sample extraction from RLWE ciphertexts. SE is a distinct kernel that bridges different ciphertext representations by converting an RLWE ciphertext into one or more LWE ciphertexts. This capability is essential not only for TFHE bootstrapping but also for cross-scheme workflows (e.g., CKKS-TFHE) that require interoperability between RLWE- and LWE-based computations. We therefore include SE as a dedicated CMK primitive for ciphertext representation conversion.

5) *Keyswitch.* The key-switching operation restores the encryption key used during EProduct to the original key domain. Functionally, the key-switching operation can be viewed as a homomorphic transformation of the LWE ciphertext ct' from one secret key to another. The core computation consists of scalar multiplications between the LWE mask components (i.e., a'_i) of ct' and the key-switching key (KSK), which comprises Nl LWE ciphertexts. TFHE Keyswitch is an arithmetic composition kernel dominated by repeated modular multiply-accumulate operations over LWE components. It introduces no additional primitives beyond modular addition and modular multiplication. Therefore, TFHE Keyswitch can be expressed using the CMK primitives MA and MM.

6) *PBS.* PBS is a composite operation that integrates all TFHE procedures described above. As such, PBS orchestrates the previously introduced TFHE kernels without introducing additional computational semantics. Accordingly, PBS can be represented as a composition of the CMK primitives, MA, MM, NTT, INTT, SE, and SQM.

2.1.1 *Conversion Between Different FHE Schemes*

In hybrid homomorphic computing, efficient conversion between different FHE schemes is a critical requirement for a unified HPU. CKKS-like schemes (including CKKS and BFV) employ RLWE-based ciphertexts, whereas TFHE uses the LWE ciphertext format. Although CKKS-like schemes share similar encryption mechanisms, TFHE differs significantly in both ciphertext representation and computation model. Consequently, efficient conversion between RLWE and LWE ciphertexts is essential for enabling cross-scheme computation within a unified CMK framework.

Within the CMK-driven framework, RLWE-LWE conversion is expressed as a composition of reusable kernel primitives. Specifically, the conversion from RLWE to LWE is captured by the SE primitive. Conversely, the conversion from LWE to RLWE is realized through an LWE homomorphic-decryption-style procedure using a repacking key (RPKey) encrypted in RLWE form. The instruction-level realization of these CMK primitives is introduced later in the H-ISA definition. The LWE-RLWE conversion workflow can be formulated as a sequence of modular multiply-accumulate operations combined with structured rotations. Specifically, MA and MM implement the arithmetic accumulation, NTT and INTT accelerates polynomial-domain multiplications when applicable, automorphism provides the rotation and permutation primitive, and SQM captures the required scaling and rounding operations. The use of automorphism enables the reuse of RPKey rotations across bs steps, reducing the asymptotic complexity from $O(N)$ to $O(\sqrt{N})$.

In summary, CMK primitives are reused across multiple FHE schemes as well as inter-scheme conversion processes, and are subsequently realized as H-ISA operations. With the exception of automorphism and SE, the remaining primitives are shared by both

CKKS-like schemes and TFHE and are also reused during ciphertext conversion between schemes. Automorphism primarily supports structured rotations and mappings in CKKS-like workflows, whereas SE bridges the RLWE and LWE ciphertext domains to enable cross-scheme computation. This CMK-level unification establishes a common architectural substrate, allowing HPU to efficiently support multi-scheme workloads by mapping shared primitives to reusable homomorphic instruction units (HIUs) under a unified abstraction.

2.2 Design Opportunities

2.2.1 *Kernel Reuse via CMK*

Although different FHE schemes appear complex and hardware-intensive, their computations can be reduced to a small set of shared kernels. This observation motivates a unified HPU that supports multiple schemes through the reuse of a compact CMK. As analyzed in Subsection 2.1, the major operations in CKKS-like schemes and TFHE can be expressed as compositions of a small set of CMK primitives, including MA, MM, NTT, automorphism, SQM, and the representation-bridging primitive SE. This commonality not only enables hardware reuse but also directly informs the design of H-ISA: each CMK primitive is mapped to a stable, reusable H-ISA operation, enabling cross-scheme composition through a unified programming interface. Guided by this abstraction, HPU instantiates a small set of reusable execution blocks that implement these H-ISA operations, rather than provisioning dedicated datapaths for individual FHE procedures. These blocks are time-multiplexed and dynamically composed through instruction scheduling, allowing HPU to support multi-scheme functionality within constrained area and bandwidth budgets.

2.2.2 *H-ISA Aware Microarchitectural Specialization*

CMK primitives exhibit highly diverse computational characteristics and execution latencies. For example, MA is a lightweight operation, whereas NTT is dominated by modular multiplications. Automorphism performs index-based permutations over large ciphertext vectors with irregular memory-access patterns, while SQM must support quotient-related operations to ensure cross-scheme compatibility. This heterogeneity can lead to pipeline imbalance if executed naively; however, it also identifies opportunities for targeted microarchitectural specialization within HPU. Accordingly, HPU introduces CMK-specialized, H-ISA aware optimizations to improve the throughput and utilization of the corresponding instruction classes. For NTT, HPU exploits the structured recurrence of twiddle factors and merges computation stages through NTT-fusion, thereby reducing the number of modular multiplications. For automorphism, HPU employs a subvector-partitioning strategy that transforms fine-grained index mappings into coarser-grained inter-subvector permutations, improving parallel execution efficiency. For SQM, HPU adopts an optimized approximate-division technique to accelerate joint quotient and modular-reduction computations while maintaining controlled precision. In summary, CMK-driven reuse determines which operations are exposed through H-ISA and implemented as reusable HIUs, whereas CMK-level specialization determines how these H-ISA operations are efficiently realized within the processor.

3 Method

The extraction of CMK primitives provides the foundation for hardware reuse and H-ISA definition. However, their computational and memory-access characteristics are often not hardware-friendly. For example, the large number of modular operations in NTT and the irregular memory-access patterns of automorphisms present significant challenges for efficient parallel execution. To address these issues, we adopt the optimization techniques proposed in Poseidon[15]. In addition, modular reduction and integer division operations are widely used across different FHE schemes and exhibit substantial computational overlap. Although modular reduction and division are mathematically distinct operations, they are closely related, as expressed by $a \mod q = a - \lfloor a/q \rfloor \times q$. This relationship motivates the design of a unified algorithm efficiently supporting both operations. Specifically, by leveraging an optimized approximate-division algorithm and refining the approximation during correction, the exact quotient and remainder can be obtained simultaneously, as shown in Algorithm 1.

As shown in Algorithm 1, the division operation can be eliminated by introducing the precomputed constant μ, which is used to obtain an approximate quotient q (line 1). The approximation error of q is bounded by 3, as proved in [16]. A straightforward correction approach is to compute $a - q \times p$ and compare the result with p. However, this method re-

Algorithm 1. SQM

Input : $k = \lfloor \log_2 p \rfloor + 1$, $0 \le a < 2^{2k}$, $\mu = \lfloor 2^{2k}/p \rfloor$.
Output: $\lfloor a/p \rfloor$, $a \bmod p$.

1 $q \leftarrow \lfloor \lfloor a/2^{k-1} \rfloor \times \mu / 2^{k+1} \rfloor$
2 $r \leftarrow (a \bmod 2^{k+1}) - (q \times p \bmod 2^{k+1})$
3 **if** $r < 0$ **then**
4 $\quad r \leftarrow r + 2^{k+1}$
5 **while** $r \ge p$ **do**
6 $\quad r \leftarrow r - p$
7 $\quad q \leftarrow q + 1$
8 **return** q, r

quires arithmetic on operands with a width of up to $2k$ bits, making it inefficient for hardware implementation. Instead, given that $0 \le a - q \times p < 3p$ and $\log_2 p < k$, it follows that $0 \le \log_2 a - q \times p \le k + 1$. Therefore, only the lower $k+1$ bits of both a and $p \times q$ need to be retained for the computation. During the subtraction in line 2 of Algorithm 1, a borrow may propagate beyond the retained bit range, resulting in a negative value of r. This condition can be corrected through a single adjustment step. Finally, iterative comparisons between r and p eliminate the residual approximation error while dynamically updating both q and r, yielding the exact quotient and remainder.

4 HPU ISA

4.1 Design Overview

To overcome the limitations of the conventional coprocessor offload model, we implement HPU as a RISC-V extension and define a dedicated H-ISA. H-ISA follows a CMK-to-ISA design principle: a CMK is first abstracted as a scheme-agnostic semantic foundation, and the resulting CMK primitives are then realized as executable instructions. Specifically, H-ISA consists of two layers: 1) micro-instructions that directly implement CMK primitives and system-level services (data movement, synchronization, and configuration); and 2) macro-instructions that provide procedure-level operations aligned with mainstream FHE programming frameworks (e.g., SEAL-style CKKS workflows and TFHE workflows), while ultimately being executed through the same underlying micro-instruction units.

H-ISA is encoded using RISC-V custom extensions. Micro-instructions are mapped to the `Custom-0` and `Custom-1` opcode spaces, whereas macro-instructions are mapped to `Custom-2`. Across all instruction classes, `rs1`, `rs2`, and `rs3` are used to specify either addresses (e.g., ciphertexts, polynomials, lookup tables, or evaluation keys) or scalar parameters, while `rd` specifies the destination address or return value, as defined in Table 1 and Table 2.

4.2 ISA Definition

4.2.1 Micro-Instructions

Micro-instructions constitute the fundamental executable interface of HPU, encompassing CMK computation primitives, data movement and synchronization, as well as configuration and status management. They are sufficient to express the CMK primitives required by FHE workloads and form the common exe-

Table 1. H-ISA Micro-Instructions of HPU

Type	Instruction (Micro)	Extension	31–25 funct7	24–20 rs2	19–15 rs1	14–12 funct3	11–7 rd	6–0 Opcode	Functional Description
Computation	`fhe.add`	`Custom-0`	0000000	rs2	rs1	100	rd	0001011	Poly addition
	`fhe.sub`	`Custom-0`	0000001	rs2	rs1	100	rd	0001011	Poly subtraction
	`fhe.mult`	`Custom-0`	0000010	rs2	rs1	100	rd	0001011	Poly multiplication
	`fhe.smult`	`Custom-0`	0000011	rs2	rs1	100	rd	0001011	Scalar poly multiplication
	`fhe.ntt`	`Custom-0`	0000100	x0	rs1	100	rd	0001011	NTT
	`fhe.intt`	`Custom-0`	0000101	x0	rs1	100	rd	0001011	Inverse NTT
	`fhe.auto`	`Custom-0`	0000110	x0	rs1	100	rd	0001011	Automorphism
	`fhe.se`	`Custom-0`	0000111	x0	rs1	100	rd	0001011	Sample extraction
Data movement	`fhe.load`	`Custom-0`	0001000	x0	rs1	100	rd	0001011	Load polynomials into scratchpad
	`fhe.store`	`Custom-0`	0001001	x0	rs1	100	rd	0001011	Store polynomials in main memory
	`fhe.fence`	`Custom-0`	0001010	x0	x0	100	x0	0001011	Synchronization
Configuration & control	`fhe.cfg`	`Custom-1`	0000000	rs2	rs1	100	x0	0101011	Config FHE parameters
	`fhe.status`	`Custom-1`	0000001	x0	rs1	100	x0	0101011	Get the status register

Table 2. H-ISA Macro-Instructions of HPU

Type	Instruction (Macro)	Extension	31–27 funct5	26–22 rs3	21–17 rs2	16–12 rs1	11–7 rd	6–0 Opcode	Functional Description
CKKS-like	`fhe.modup`	`Custom-2`	00000	x0	rs2	rs1	rd	1 011 011	Modulus raise
	`fhe.moddown`	`Custom-2`	00001	x0	rs2	rs1	rd	1 011 011	Modulus reduce
	`fhe.pmult`	`Custom-2`	00010	x0	rs2	rs1	rd	1 011 011	Plaintext multiplication
	`fhe.smult`	`Custom-2`	00011	x0	rs2	rs1	rd	1 011 011	Scalar multiplication
	`fhe.cmult`	`Custom-2`	00100	rs3	rs2	rs1	rd	1 011 011	Ciphertext multiplication
	`fhe.hadd`	`Custom-2`	00101	x0	rs2	rs1	rd	1 011 011	Homomorphic addition
	`fhe.hsub`	`Custom-2`	00110	x0	rs2	rs1	rd	1 011 011	Homomorphic subtraction
	`fhe.rotate`	`Custom-2`	00111	rs3	rs2	rs1	rd	1 011 011	Rotation
	`fhe.ks`	`Custom-2`	01000	rs3	rs2	rs1	rd	1 011 011	Key switching
	`fhe.relin`	`Custom-2`	01001	rs3	rs2	rs1	rd	1 011 011	Relinearization
TFHE	`fhe.cmux`	`Custom-2`	01010	rs3	rs2	rs1	rd	1 011 011	Homomorphic CMUX gate
	`fhe.lwe.ks`	`Custom-2`	01011	x0	rs2	rs1	rd	1 011 011	Key switching
	`fhe.lwe.hadd`	`Custom-2`	01100	x0	rs2	rs1	rd	1 011 011	Homomorphic addition
	`fhe.lwe.sub`	`Custom-2`	01101	x0	rs2	rs1	rd	1 011 011	Homomorphic subtraction
	`fhe.lwe.pmult`	`Custom-2`	01110	x0	rs2	rs1	rd	1 011 011	Plaintext multiplication

cution substrate upon which macro-instructions are built.

Computation Micro-Instructions (`Custom-0`). We define the following computation micro-instructions: `fhe.add` (polynomial addition), `fhe.sub` (polynomial subtraction), `fhe.mult` (polynomial multiplication), `fhe.smult` (scalar-polynomial multiplication), `fhe.ntt` (forward NTT), `fhe.intt` (inverse NTT), `fhe.auto` (automorphism), and `fhe.se` (sample extraction). For `fhe.add`, `fhe.sub`, and `fhe.mult`, `rs1` and `rs2` specify the source addresses, while `rd` specifies the destination address. For `fhe.smult`, `rs1` specifies the source address, `rs2` provides the scalar operand, and `rd` specifies the destination address. For transform and permutation operations (`fhe.ntt`, `fhe.intt`, `fhe.auto`, and `fhe.se`), `rs1` specifies the source address and `rd` specifies the destination address, whereas unused operand fields are set to `x0`. These computation micro-instructions directly implement the CMK primitives, including modular arithmetic kernels, domain-transform operations, and structured permutation and representation-conversion kernels.

Data Movement and Synchronization (`Custom-0`). HPU employs an explicit scratchpad memory to stage ciphertext and polynomial data, thereby reducing external memory traffic. Accordingly, we define `fhe.load` and `fhe.store` to transfer polynomials between main memory and the scratchpad. For both instructions, `rs1` specifies the virtual address in main memory, while the destination or source scratchpad address is encoded in $\texttt{rd2}[25:0]$. The field $\texttt{rd2}[31:26]$ encodes the number of polynomials to be transferred. We further define `fhe.fence` as a synchronization primitive that enforces ordering between data-movement and computation operations, enabling deterministic scheduling and the safe composition of pipelined execution flows.

Configuration and Status (`Custom-1`). To support multiple FHE schemes under a unified interface, HPU exposes scheme-specific parameters through `fhe.cfg`. This instruction configures, at a minimum, the polynomial degree, decomposition parameters, and maximum modulus-chain information for both RLWE- and LWE-based ciphertext representations. The status of long-latency operations can be queried through `fhe.status`, which returns a running or completed indicator in `rs1`. Together, these instructions enable the software stack to configure, monitor, and orchestrate multi-scheme execution without embedding scheme-specific behavior directly into the instruction stream.

4.2.2 Macro-Instructions

Macro-instructions provide a procedure-level abstraction aligned with mainstream FHE programming frameworks while preserving the unifying benefits of CMK. Conceptually, each macro-instruction can be viewed as a semantic bundle that 1) reuses the same underlying HIUs as micro-instructions, and 2) internally orchestrates a short sequence of CMK primitives. This design improves programmability and reduces instruction-stream overhead for high-level FHE operations while maintaining execution on a common reusable kernel substrate. Macro-instructions are encoded within `Custom-2` and include an additional operand, `rs3`, for passing auxiliary addresses

(e.g., NTT tables or evaluation keys) when required. The `rd` operand specifies the destination address.

CKKS-Like Macro-Instructions. We define the following CKKS-like macro-instructions: `fhe.modup` (modulus raising), `fhe.moddown` (modulus reduction), `fhe.pmult` (plaintext multiplication), `fhe.smult` (scalar multiplication), `fhe.cmult` (ciphertext multiplication), `fhe.hadd` (homomorphic addition), `fhe.hsub` (homomorphic subtraction), `fhe.rotate` (rotation), `fhe.ks` (key switching), and `fhe.relin` (relinearization). These macro-instructions receive ciphertext or polynomial source addresses through `rs1` and `rs2`, write results to `rd`, and optionally consume auxiliary tables or keys. Specifically, `rs2` or `rs3` may provide the NTT-table address, while `rs3` may provide the evaluation-key address (e.g., for rotation, key switching, or relinearization). Although these operations appear scheme-specific at the algorithmic level, their execution is realized through the composition of CMK primitives and the corresponding H-ISA micro-instructions, including `fhe.ntt` and `fhe.intt`.

TFHE Macro-Instructions. We define the following TFHE-oriented macro-instructions: `fhe.cmux` (homomorphic controlled multiplexer (CMUX) gate), `fhe.lwe.ks` (LWE key switching), `fhe.lwe.hadd` (LWE homomorphic addition), `fhe.lwe.sub` (LWE homomorphic subtraction), and `fhe.lwe.pmult` (plaintext multiplication). These instructions follow the same `rs1`/`rs2`/`rs3`-to-`rd` operand convention as the CKKS-like macro-instructions. Auxiliary inputs, such as key addresses required for TFHE key switching, are supplied through the designated operand registers according to the format defined in Table 2. Importantly, TFHE macro-instructions reuse the same underlying CMK primitives as CKKS-like workflows whenever applicable, while exposing TFHE-specific procedures through a framework-oriented programming interface.

Macro-Instructions as CMK-Preserving Abstractions. Macro-instructions do not introduce computation semantics beyond those defined by the CMK. Instead, they encapsulate frequently used high-level FHE procedures to improve programmability and reduce software overhead. Consequently, HPU can schedule and execute both multi-scheme and cross-scheme workloads on a unified architectural substrate, where micro- and macro-instructions ultimately map to a shared pool of HIUs that implement the CMK primitives.

4.3 Completeness Proof of H-ISA

We prove that H-ISA is complete for the targeted class of multi-scheme FHE workloads and remains compatible with the RISC-V programming model. Completeness implies that every CMK primitive required by CKKS-like and TFHE operations can be realized through a finite H-ISA micro-program, whereas macro-instructions serve only as CMK-preserving abstractions. Compatibility implies that H-ISA extends rather than replaces the base ISA: general-purpose control flow, integer computation, and memory management remain expressible through standard RISC-V instructions, while H-ISA introduces only homomorphic-specific semantics.

4.3.1 Definitions

Definition 1 (Architectural State). *Let Σ denote the HPU architectural state, including 1) scratchpad- and high bandwidth memory (HBM)-resident operand regions, 2) scheme- and parameter-configuration registers, and 3) control and status registers.*

Definition 2 (CMK). *Let $\mathcal{K}$ denote the CMK primitive set defined in Subsection 2.1. Each $k \in \mathcal{K}$ represents a (possibly partial) state-transition function $k : \Sigma \to \Sigma$ under a fixed parameter configuration.*

Definition 3 (Micro-Instruction Semantics). *Let $\mathcal{I}_\mu$ denote the set of micro-instructions defined in Table 1. Each $i \in \mathcal{I}_\mu$ induces a deterministic state-transition function $i : \Sigma \to \Sigma$. For a finite micro-program $\pi = (i_1, \ldots, i_T)$, define $\pi = i_T \circ \cdots \circ i_1$.*

Definition 4 (Expressibility). *A CMK primitive $k \in \mathcal{K}$ is expressible by H-ISA if there exists a finite micro-instruction sequence $\pi(k)$ such that $\pi(k)(\sigma) = k(\sigma)$ for all $\sigma \in \Sigma$ satisfying the preconditions of k.*

4.3.2 Key Lemmas

Lemma 1 (CMK Realization by Micro-Instructions). *For every CMK primitive $k \in \mathcal{K}$ required by the targeted CKKS-like and TFHE workloads considered in this study, k is expressible by H-ISA micro-instructions.*

Proof. The claim follows by direct construction. Arithmetic primitives are realized by `fhe.add`, `fhe.sub`, `fhe.mult`, and `fhe.smult`; domain-transform primitives are realized by `fhe.ntt` and `fhe.intt`; and structured permutation and representation-conversion primitives are realized by `fhe.auto` and `fhe.se`,

respectively. Operand movement and execution ordering are provided by `fhe.load`, `fhe.store`, and `fhe.fence`. Configuration and execution-status management are supported by `fhe.cfg` and `fhe.status`. Therefore, every CMK primitive k admits a finite realizing micro-program $\pi(k)$. □

Lemma 2 (Macro-Instructions Are CMK-Preserving). *Let $\mathcal{I}_M$ denote the macro-instruction set defined in Table 2. For every $I \in \mathcal{I}_M$, there exists a finite sequence of CMK primitives $(k_1, \ldots, k_m)$ such that $I = k_m \circ \ldots \circ k_1$ for all architectural states on which I is defined. Consequently, I introduces no computational semantics beyond those of the CMK.*

Proof. Each macro-instruction corresponds to a standard FHE procedure whose algorithmic workflow can be decomposed into a finite composition of the CMK primitives defined in Subsection 2.1. The macro-instruction interface merely encapsulates this composition while supplying auxiliary operands (lookup tables and evaluation keys) through architectural registers. □

Lemma 3 (RISC-V Compatibility and Whole-Program Expressibility). *Let $\mathcal{I}_R$ denote the base RISC-V ISA executed by the host core. Any targeted multi-scheme FHE workload can be expressed as a program over $\mathcal{I}_R \cup \mathcal{I}_\mu$ (or equivalently $\mathcal{I}_R \cup \mathcal{I}_M$), in which all non-homomorphic computation and control flow are implemented using $\mathcal{I}_R$, while homomorphic kernels are implemented using H-ISA.*

Proof. The targeted workloads consist of two components: general-purpose host-side logic and homomorphic computation. The former includes loops, branching, address generation, framework-level function invocation, and memory management, all of which are directly expressible using $\mathcal{I}_R$. The latter reduces to compositions of CMK primitives and is therefore expressible by Lemma 1, or equivalently invocable through the macro-instructions characterized in Lemma 2. Since H-ISA is introduced as an extension to RISC-V ISA, it coexists with the standard instruction-fetch, decode, and execution mechanisms while preserving the semantics of the base ISA. □

4.3.3 Completeness Theorem

Theorem 1 (H-ISA Completeness with RISC-V Compatibility). *For the CKKS-like and TFHE procedures considered in this study, including CMK-based inter-scheme conversion operations, every procedure can be expressed as a finite H-ISA micro-program. Furthermore, any end-to-end workload can be expressed as a finite program over the combined instruction set $\mathcal{I}_R \cup \mathcal{I}_\mu$. Equivalent functionality may also be invoked through macro-instructions, each of which expands into a CMK-preserving micro-instruction sequence.*

Proof. Any targeted procedure P can be decomposed into a finite composition of CMK primitives. Therefore, there exist $k_1, \ldots, k_n \in \mathcal{K}$ such that $P = k_n \circ \ldots \circ k_1$ on the domain of P. By Lemma 1, for each primitive k_j, there exists a finite micro-program $\pi(k_j)$ satisfying $\pi(k_j) = k_j$. By concatenating these micro-programs, we obtain a finite H-ISA micro-program $\pi(P)$ such that $\pi(P) = P$. Therefore, every targeted procedure is expressible through H-ISA micro-instructions. By Lemma 2, macro-instructions are CMK-preserving abstractions and thus provide equivalent functionality without introducing additional computational semantics. Finally, Lemma 3 establishes that the combination of the base RISC-V ISA for general-purpose computation and H-ISA for homomorphic kernels provides whole-program expressibility while preserving the semantics of the base ISA. Hence, the theorem follows. □

5 HPU Architecture

5.1 Overview

Fig.1 shows the HPU architecture and a programming example. HPU extends a RISC-V core via rocket custom coprocessor (RoCC) and exposes H-ISA as custom instructions. Homomorphic execution is ISA-driven: instructions are buffered, dependency-checked, decoded, and issued to either data movement or execution units, enabling instruction-level scheduling and overlap between transfers and compute across CKKS-like and TFHE workloads. The front-end comprises an instruction buffer, dependency management, a decoder, and a multi-issue controller. The decoder classifies incoming commands into `Load`, `Store`, and `EXE`; the controller issues them concurrently when operands and destinations do not conflict. The memory subsystem centers on an on-chip static random-access memory (SRAM) scratchpad for ciphertext or polynomial operands and intermediates. To improve SRAM utilization, HPU integrates a local memory management unit (MMU) with a local translation lookaside buffer (TLB) and manages scratchpad storage at polynomial granularity using a page-based homomorphic mech-

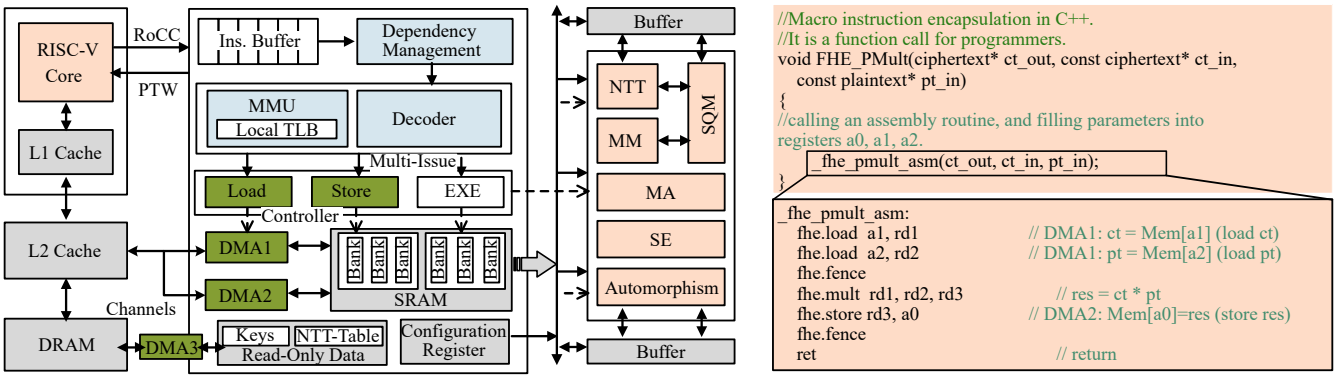


Fig.1. HPU architecture overview and programming example. (a) A RISC-V RoCC extension routes H-ISA instructions through decoding and dependency-aware scheduling to shared HIUs, with a local homomorphic MMU and TLB managing polynomial-granular scratchpad addressing and dedicated double data rate (DDR) channels streaming read-only evaluation keys (EVKs) and NTT tables. (b) A framework call is lowered to H-ISA via inline assembly, where macro-instructions are decomposed into micro-instructions and then executed on HIUs. Ins. means homomorphic instruction unit.

anism (proposed in [17]). This computation-storage co-design improves data locality and efficiency, ensuring better utilization of the scratchpad while supporting homomorphic operations across varying ciphertext states. After decoding, `fhe.load` and `fhe.store` generate virtual addresses for scratchpad access, which are translated into physical addresses by the homomorphic MMU. Direct memory access (DMA) engines stream data between L2 cache or dynamic random-access memory (DRAM) and the scratchpad, while `fhe.fence` enforces ordering between transfers and execution.

The execution back-end comprises HIUs that implement CMK primitives exposed by micro-instructions (e.g., MA, MM, NTT, INTT, automorphism, SE, and SQM support). In addition, large read-only datasets such as evaluation keys (`evk`) and NTT tables are accessed via a dedicated DDR channel as read-only streams, providing higher bandwidth and a predictable data supply to HIUs without cache pollution.

Fig.1(b) illustrates the programming structure using `PMult`. At the top level, programmers write C++ code in a conventional framework style, where a function acts as a macro-instruction wrapper and defines a stable call interface. Inside the wrapper, inline assembly materializes the HPU invocation by passing operands through registers and issuing the corresponding macro-instruction (e.g., `fhe.pmult`). The macro-instruction serves as a pseudoinstruction at the ISA level: it represents a single semantic operation for programmers and compilers, while the HPU expands it into a sequence of micro-instructions that drive hardware resources. In the `PMult` example, this expansion includes 1) `fhe.load` to stage operands, 2) `fhe.fence` to synchronize DMA and execution, 3) compute micro-instructions such as `fhe.mult`, and 4) `fhe.store` to commit results. This structure makes the programming model explicit: C++ framework calls map to macro-level semantics, while performance-critical execution is driven by micro-instructions that directly target HIUs and the scratchpad-managed pipeline.

5.2 HIUs

HPU instantiates six HIU types, each implementing one or more primitive instructions in H-ISA: MA, MM, NTT, automorphism, SE, and SQM. MA realizes modular addition in the finite ring (i.e., comparison and subtraction under modulus q). MM realizes modular multiplication and leverages SQM for reduction. Since existing optimized designs from Poseidon are adopted for NTT and automorphism, this section focuses on the design of SE and SQM.

5.2.1 *SQM HIU*

SQM HIU executes division and modulus operations within a unified instruction framework, following the algorithm introduced in Section 3. The hardware implementation is shown in Fig.2. The higher $(k+1)$ bits of the product of input operands a and μ provide an approximate quotient q. Subtracting the product $a \times (q \times p)$ yields an approximate remainder. This HIU requires only two narrow-width multipliers; subsequent operations use adders and subtractors, forming a fully pipelined datapath. The loop structure is unfolded, enabling one-result-per-cycle throughput.

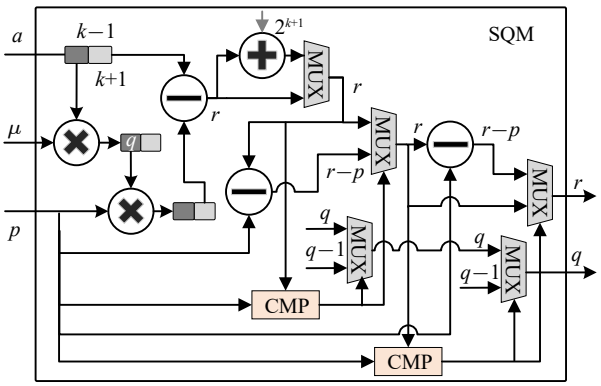


Fig.2. SQM HIU microarchitecture.

5.2.2 SE HIU

Sample extraction enables ciphertext transformation between different FHE schemes, primarily supporting conversion from RLWE to LWE formats. Although computationally lightweight, it incurs significant data movement. To extract $ct_i(lwe)$, coefficients are cyclically shifted by $(i+1)$ positions, and elements from $(i+1)$ to N are negated. While barrel shifters can realize $O(\log N)$ shifts, this approach becomes inefficient for repeated sequential extractions. In hybrid FHE applications, a single sample extraction may require up to N extractions. Therefore, we implement SE as a cyclic-shift HIU. As shown in Fig.3, ciphertext segments are distributed across block random-access memory (BRAM) based FIFOs (first in first outs) in a head-to-tail topology. Each cycle performs a one-position shift, while negations are handled by subtractors controlled via sign-selection logic. Due to limited FIFO bandwidth, data for multiple rows are read in parallel, leading to temporary inconsistencies between row- and column-major layouts.

6 Evaluation

6.1 Experimental Setup

Platform. HPU is implemented as a RISC-V processor extension that executes homomorphic workloads through the proposed H-ISA and integrated HIUs. We prototype the HPU on a Xilinx Alveo U280 FPGA operating at 300 MHz and connected to an x86 host through PCIe. Vivado and Vitis 2021.1 and the Xilinx Runtime are used for system configuration and workload execution. To validate the processor-style integration model, we deploy a lightweight open-source Rocket core configured as RV32G and integrate H-ISA decoding and HIUs through the RoCC interface. The core adopts a minimal single-issue microarchitecture to reduce FPGA resource overhead while preserving full instruction-level control. The FPGA memory hierarchy comprises a 128 KB L1 cache and a 4 MB L2 cache implemented using BRAM and ultra RAM (URAM) resources. Large ciphertexts and intermediate data are stored in the on-board HBM, which serves as the primary high-capacity memory instead of conventional DDR memory.

Baseline. We use a single-threaded Intel Xeon Silver 4144 CPU operating at 2.2 GHz as the software baseline, with SEAL for CKKS-like workloads and TFHE for TFHE workloads. For CKKS-like applications, we compare HPU against FPGA-based and GPU-based Over100x[18], HEAX[19], FAB[20], and TensorFHE[21], and ASIC-based CraterLake[22], BTS[23], ARK[24], and SHARP[25]. For TFHE workloads, we compare HPU against the GPU-based implementations NuFHE① and CuFHE[26], the FPGA-based accelerator XHEC[27], and the ASIC designs Strix[28] and Morphling[29]. To evaluate heterogeneous FHE sup-

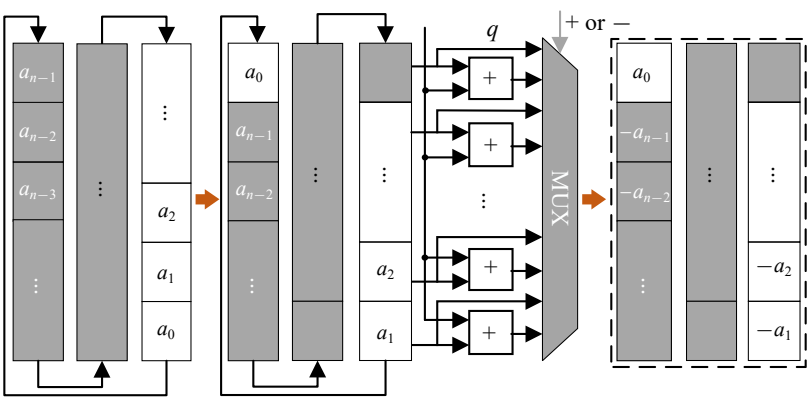


Fig.3. SE HIU microarchitecture.

①https://github.com/nucypher/nufhe, May 2026.

port, we further compare HPU with multi-scheme accelerators, including Trinity[13] and UFC[14]. Trinity and UFC represent prior efforts to support multiple FHE schemes within a unified hardware framework.

Benchmarks. We evaluate HPU using the following four benchmarks.

1) *ResNet-20:* a CKKS-based encrypted convolutional neural network (CNN) trained on CIFAR-10, following the implementation in [2]. The model comprises 19 convolutional layers and one fully connected layer.

2) *LSTM:* a TFHE-based long short-term memory (LSTM) network that uses programmable bootstrapping to evaluate nonlinear functions, following the implementation in [10]. The workload processes 128 encrypted LWE ciphertexts sequentially.

3) *PEGASUS:* a hybrid CKKS-TFHE implementation of K-means clustering based on PEGASUS[11], which exploits CKKS SIMD parallelism for linear computations and TFHE PBS for nonlinear operations.

4) *AFBoot:* a hybrid BFV-TFHE batch bootstrapping scheme proposed in [12], which packs multiple LWE ciphertexts into BFV ciphertexts and evaluates lookup-table functions. Together, these benchmarks span arithmetic, Boolean, and hybrid homomorphic workloads, enabling a comprehensive evaluation of HPU across diverse FHE schemes, computation models, and application scenarios.

6.2 Performance

6.2.1 Basic FHE Operations

The performance of basic FHE operations is a primary determinant of end-to-end application efficiency. In the HPU prototype, these operations are executed through the proposed RISC-V H-ISA extension: each operation is translated into H-ISA micro- or macro-instructions, issued by the HPU front-end, and executed on the corresponding HIUs using scratchpad-resident operands. Table 3 reports the throughput of basic CKKS-like operations (CKKS and BFV), including `PMult`, `CMult`, `HAdd`, and `Rotation`. The CKKS parameters follow [18], with $N=2^{16}$ and $\log Q=1\,693$. The BFV parameters are $N=65\,536$, $\log Q=1\,639$, and $\log t=36$, where t denotes the plaintext modulus. We use the number of operations performed per second as the metric. Across all operators, HPU achieves throughput improvements of two to three orders of magnitude over the CPU baseline. Compared with the GPU-based accelerator Over100x, which targets CKKS workloads, HPU delivers several-fold higher throughput. HPU also outperforms FPGA-based accelerators such as HEAX[19] and FAB[20]. A key reason is its instruction-driven execution model. HEAX dedicates substantial hardware resources to a fully pipelined `Keyswitch` implementation, which may result in resource underutilization under mixed-operator workloads. In contrast, HPU dynamically reuses a shared pool of HIUs across operations such as `CMult`, `Keyswitch`, `Rotation`, and `Rescale` through instruction-level scheduling and resource sharing. For lightweight operators, including `HAdd` and `PMult`, HPU achieves throughput comparable to that of FAB because both architectures provide similar modular-arithmetic capacity and operate at comparable clock frequencies.

For compute-intensive operators dominated by NTT-based transforms (`CMult`, `Keyswitch`, `Rotation`, `Rescale`), HPU further benefits from its fused NTT pipeline and flexible H-ISA level instruction scheduling, achieving a 1.16x–1.56x throughput improvement over FAB. A similar trend is observed for BFV operators, as CKKS and BFV share the same CMK-level execution patterns under the unified H-ISA abstraction. Throughout the evaluation, throughput is measured in terms of operations per second.

6.2.2 Full-System Performance

This experiment evaluates the end-to-end performance of HPU against representative GPU, FPGA, and ASIC prototypes. Because the benchmarks span

Table 3. Performance Comparison of Basic FHE Operations

Platform	CKKS						BFV			
	HAdd	PMult	CMult	Keyswitch	Rotation	Rescale	HAdd	PMult	CMult	Rotation
CPU	57.01	37.45	0.48	0.49	0.52	8.34	57.01	37.45	0.15	0.26
Over100x (GPU)[18]	6 250.00	7 407.00	337.00	/	392.00	2 040.00	/	/	/	/
HEAX (FPGA)[19]	4 161.00	4 161.00	119.00	104.00	/	/	/	/	/	/
FAB (FPGA)[20]	25 000.00	25 000.00	584.00	617.00	636.00	5 263.00	25 000.00	25 000.00	190.00	2 632.00
HPU (FPGA)	25 000.00	25 000.00	741.00	782.00	735.00	8 217.00	25 000.00	25 000.00	241.00	4 109.00

multiple FHE schemes, most existing scheme-specific accelerators support only a subset of workloads; in particular, none of the prior baselines supports PEGASUS[11], which requires cross-scheme execution. As shown in Table 4, for the CKKS-based ResNet-20 workload, HPU outperforms two GPU-based accelerators by 3.65x and 1.91x, respectively, and achieves a 1.36x speedup over the SOTA FPGA accelerator FAB. This gain primarily stems from instruction-driven resource multiplexing and scheduling, which sustains higher throughput in key kernels (e.g., NTT and automorphism) without requiring dedicated fixed-function units per operation. The FPGA-based HPU prototype is approximately an order of magnitude slower than most ASIC accelerators due to lower operating frequency and more constrained compute and memory resources, but performs comparably to BTS[23], likely due to suboptimal software-hardware parameter matching in BTS. We further evaluate a 7 nm ASIC implementation of HPU in Subsection 6.3. For the TFHE-based LSTM workload, HPU outperforms both GPU and FPGA accelerators, achieving a 282.73x speedup over NuFHE and a 1.20x improvement over XHEC[29]. Finally, for the multi-scheme PEGASUS[11] and AFBoot[12] workloads (unsupported by prior accelerators), HPU achieves 766x and 883x speedups over the CPU baseline, respectively.

6.3 ASIC Implementation

To validate the scalability and physical feasibility of HPU, we implement the full design in register-transfer level (RTL) using the ASAP7 7.5-track 7 nm predictive PDK[30]. SRAM structures are modeled using FinCACTI[31], and the area and power of two HBM stacks are estimated based on prior characterization studies[32]. The ASIC-based HPU operates at 1 GHz and is architecturally consistent with the FPGA prototype: it preserves the same H-ISA driven microarchitecture and instruction semantics, differing primarily in that the richer silicon budget enables higher parallelism and larger on-chip storage. Specifically, HIU data-level parallelism is scaled from 512 to 4 096 lanes, and two HBM stacks are integrated to satisfy the increased bandwidth demand, providing a theoretical peak off-chip bandwidth of 1 TB/s. To improve data reuse under higher parallelism, we incorporate an on-chip cache subsystem. Since CKKS-like workloads typically exhibit larger working sets than TFHE, we adopt the cache optimization strategy from MAD[33] to reduce capacity overhead in large-parameter configurations. In the final design, a 32 MB on-chip cache provides an aggregate bandwidth of approximately 36 TB/s, supplying data to the 4 096-lane HIU pipeline.

Table 5 summarizes the area and power breakdown. The ASIC-equivalent HPU occupies 111.76 mm^2 and consumes 101.03 W at 1 GHz, including a lightweight Rocket RV32G control core (2.4 mm^2, 1.8 W) used for RoCC and H-ISA integration and instruction-level control. The Rocket core overhead is modest, and the overall area and power budgets are dominated by the data path and memory subsystem. Memory structures (cache and HBM interfaces) dominate both area and power consumption, consistent with the data-intensive nature of FHE workloads. Among HIUs, the NTT unit is the most resource-intensive due to its computational complexity and its dominant role across workloads. Overall, the 7 nm results confirm that HPU preserves a consistent architecture from FPGA to ASIC while scaling performance through increased HIU parallelism and expanded cache capacity.

Performance. Since Trinity and UFC target dif-

Table 4. Full-System Latency (ms) Comparison with SOTA Accelerator Prototypes

Application	ResNet-20 (CKKS)	LSTM (TFHE)	PEGASUS (CKKS&TFHE)	AFBoot (BFV&TFHE)
Over100x (GPU)	9 420.00	/	/	/
TensorFHE (GPU)	4 939.00	/	/	/
FAB (FPGA)	3 510.00	/	/	/
CraterLake (ASIC)	249.45	/	/	/
BTS-1 (ASIC)	1 910.00	/	/	/
ARK (ASIC)	125.00	/	/	/
SHARP (ASIC)	99.00	/	/	/
NuFHE (GPU)	/	107 520.00	/	/
XHEC (FPGA)	/	457.10	/	/
Strix (ASIC)	/	58.90	/	/
Morphling (ASIC)	/	51.20	/	/
HPU (FPGA)	2 582.00	380.29	425.64	1 084.35

Table 5. Area and Power Breakdown (1 GHz, 7 nm)

Component	Area (mm^2)	Power (W)
Automorphism	1.04	0.78
MA	0.56	1.18
MM	4.84	11.60
NTT	51.20	40.20
SE	0.72	1.96
NoC	7.10	8.30
Scratchpad SRAM (32 MB)	14.30	3.41
HBM (2x HBM2E) (Capacity 16 GB / Bandwidth 1 TB/s)	29.60	31.80
Sum (7 nm)	111.76	101.03
CraterLake (7 nm)	222.70	∼207.00
SHARP (7 nm)	178.80	/
Strix (28 nm)	141.37	77.14
Morphling (28 nm)	74.79	53.00
Trinity (7 nm)	157.26	229.36
UFC (7 nm)	197.70	76.90

ferent application suites and overlap with HPU only on ResNet-20, we evaluate their performance using the cycle-level simulation framework from MAD[33]. Compared with CKKS-oriented ASIC accelerators (e.g., SHARP[25] and CraterLake[22]), HPU achieves comparable ResNet-20 performance while improving area and power efficiency. The main contributors are instruction-driven multiplexing across shared HIUs, which improves utilization without requiring dedicated fixed-function pipelines, and a cache policy (following MAD[33]) that reduces on-chip capacity requirements without degrading performance. Compared with TFHE-oriented ASIC designs (Strix and Morphling), HPU does not significantly reduce total area or power, primarily because it provisions larger SRAM and HBM subsystems to maintain balanced support for both CKKS-like and TFHE workloads. Nevertheless, HPU's compute-side overhead remains competitive, and HPU achieves higher throughput than Strix and Morphling by 1.57x and 1.37x, respectively (Table 6). This advantage stems from HPU's unified instruction-level execution: instead of relying on deeply cascaded, scheme-specialized pipelines that require abundant task-level parallelism, HPU decomposes kernels into fine-grained instructions scheduled on shared HIUs, reducing effective latency and sustaining throughput even for less-parallel workloads such as LSTM.

Table 6. ASIC Implementation Performance (ms)

Platform	ResNet-20	LSTM	PEGASUS	AFBoot
CraterLake	321.0	/	/	/
SHARP	99.0	/	/	/
Strix	/	58.9	/	/
Morphling	/	51.2	/	/
Trinity	89.0	34.8	16.1	56.3
UFC	90.0	27.5	23.9	18.4
HPU (ASIC)	107.6	19.5	15.5	44.0

Compared with the multi-scheme architecture Trinity[13], HPU demonstrates clear advantages on TFHE (LSTM) and hybrid workloads (PEGASUS and AFBoot), achieving a 1.78x speedup on LSTM. On CKKS ResNet-20, HPU is slightly slower, primarily due to its smaller silicon footprint (approximately 70% of Trinity's area). Under matched resource budgets, HPU is expected to outperform Trinity more consistently. Architecturally, HPU's advantage on TFHE stems from a more complete H-ISA abstraction and instruction-level reuse, which keeps HIUs highly utilized across schemes. In contrast, Trinity achieves cross-scheme generalization primarily by flexibly composing NTT units for different polynomial lengths. This design can lead to underutilization of NTT acceleration units in small-parameter TFHE workloads, thereby reducing throughput on TFHE and hybrid applications. Compared with UFC[14], HPU achieves higher performance on LSTM and PEGASUS (1.41x and 1.54x, respectively), whereas UFC performs better on CKKS and AFBoot, particularly AFBoot. This gap is primarily explained by UFC's substantially larger compute provisioning (up to 16 384 -way parallelism), which also increases the chip area to 1.8x that of HPU. In addition, UFC's performance is less uniform because it enforces multifunctional processing elements across operations, which can require algorithmic restructuring that introduces redundant computation. For example, to reduce routing and network on chip (NoC) overhead, UFC implements CKKS automorphism by adding extra NTT stages and replaces TFHE sample extraction with monomial-polynomial multiplication, both of which increase NTT-dominated computation. Such modifications trade hardware reuse for increased arithmetic complexity, resulting in noticeable workload-dependent performance imbalance.

This advantage primarily stems from HPU's processor-style, instruction-controlled execution. H-ISA exposes CMK primitives and conversion-supporting instructions, which are executed on dedicated HIUs and scheduled to match the homomorphic dataflow. In contrast, GPU-based designs such as BoostCom[34] execute FHE workloads on general-purpose arithmetic pipelines and memory hierarchies that are not co-designed for FHE-specific transforms and data

movement, resulting in lower efficiency on homomorphic workloads.

7 Conclusions

In this study, we presented the Homomorphic Processing Unit (HPU), a unified processing unit for accelerating fully homomorphic encryption. By replacing scheme-specific, fixed-function designs with an ISA-driven and reusable architecture, HPU provides a consistent execution substrate for arithmetic, Boolean, and hybrid workloads. It further adopts a computation-storage co-design that aligns polynomial-granular micro-instructions with polynomial-granular scratchpad management, improving on-chip storage efficiency. We prototyped HPU on an Xilinx Alveo U280 FPGA and evaluated a 7-nm ASIC implementation. The results show that HPU achieves competitive performance and energy efficiency compared with prior GPU-, FPGA-, and ASIC-based solutions while supporting broader multi-scheme workloads. Overall, HPU demonstrates that a processor-style architectural abstraction based on a complete ISA is an effective direction for practical homomorphic computing systems.

Conflict of Interest Xiao-Wei Li is an associate Editor-in-Chief for the Journal of Computer Science and Technology and was not involved in the editorial review of this article. The authors declare that there are no other competing interests.

References

[1] Chen H, Laine K, Rindal P. Fast private set intersection from homomorphic encryption. In *Proc. the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 30–Nov. 3, 2017, pp.1243–1255. DOI: 10.1145/3133956.3134061.

[2] Lee E, Lee J W, Lee J, Kim Y S, Kim Y, No J S, Choi W. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *Proc. the 39th International Conference on Machine Learning*, Jul. 2022, pp.12403–12422.

[3] Han K, Hong S, Cheon J H, Park D. Logistic regression on homomorphic encrypted data at scale. In *Proc. the 33rd AAAI Conference on Artificial Intelligence*, Jan. 27–Feb. 1, 2019, pp.9466–9471. DOI: 10.1609/aaai.v33i01.33019466.

[4] Fan J, Vercauteren F. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144. 2012. https://eprint.iacr.org/2012/144, May 2026.

[5] Brakerski Z, Gentry C, Vaikuntanathan V. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Computation Theory (TOCT)*, 2014, 6(3): Article No. 13. DOI: 10.1145/2633600.

[6] Cheon J H, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of approximate numbers. In *Proc. the 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, Dec. 2017, pp.409–437. DOI: 10.1007/978-3-319-70694-8_15.

[7] Chillotti I, Gama N, Georgieva M, Izabachene M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Proc. the 22nd International Conference on the Theory and Application of Cryptology and Information Security*, Dec. 2016, pp.3–33. DOI: 10.1007/978-3-662-53887-6_1.

[8] Shen L, Chen X, Wang D, Fang B, Dong Y. Efficient and private set intersection of human genomes. In *Proc. the 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Dec. 2018, pp.761–764. DOI: 10.1109/BIBM.2018.8621291.

[9] Hesamifard E, Takabi H, Ghasemi M. CryptoDL: Deep neural networks over encrypted data. arXiv: 1711.05189, 2017. https://arxiv.org/abs/1711.05189, Jun. 2026.

[10] Trama D, Clet P E, Boudguiga A, Sirdey R. Building blocks for LSTM homomorphic evaluation with TFHE. In *Proc. the 7th International Symposium on Cyber Security, Cryptology, and Machine Learning*, Jun. 2023, pp.117–134. DOI: 10.1007/978-3-031-34671-2_9.

[11] Lu W J, Huang Z, Hong C, Ma Y, Qu H. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *Proc. the 2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp.1057–1073. DOI: 10.1109/SP40001.2021.00043.

[12] Liu Z, Wang Y. Amortized functional bootstrapping in less than 7 ms, with $\tilde{O}(1)$ polynomial multiplications. In *Proc. the 29th International Conference on the Theory and Application of Cryptology and Information Security*, Dec. 2023, pp.101–132. DOI: 10.1007/978-981-99-8736-8_4.

[13] Deng X, Fan S, Hu Z, Tian Z, Yang Z, Yu J, Cao D, Meng D, Hou R, Li M, Lou Q, Zhang M. Trinity: A general purpose FHE accelerator. In *Proc. the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2024, pp.338–351. DOI: 10.1109/MICRO61859.2024.00033.

[14] Zhou M, Nam Y, Wang X, Lee Y, Wilkerson C, Kumar R, Taneja S, Mathew S, Cammarota R, Rosing T. UFC: A unified accelerator for fully homomorphic encryption. In *Proc. the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2024, pp.352–365. DOI: 10.1109/MICRO61859.2024.00034.

[15] Yang Y, Zhang H, Fan S, Lu H, Zhang M, Li X. Poseidon: Practical homomorphic encryption accelerator. In

Proc. the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Feb. 25–Mar. 1, 2023, pp.870–881. DOI: 10.1109/HPCA56546.2023.10070984.

[16] Hankerson D, Vanstone S, Menezes A. Guide to Elliptic Curve Cryptography. Springer, 2004. DOI: 10.1007/b97644.

[17] Wang H, Yang Y, Lu H, Li X. Hypnos: Memory efficient homomorphic processing unit. In *Proc. the 62nd ACM/IEEE Design Automation Conference (DAC)*, Jun. 2025, pp.1–7. DOI: 10.1109/DAC63849.2025.11132418.

[18] Jung W, Kim S, Ahn J H, Cheon J H, Lee Y. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Trans. Cryptographic Hardware and Embedded Systems*, 2021, 2021(4): 114–148. DOI: 10.46586/tches.v2021.i4.114-148.

[19] Riazi M S, Laine K, Pelton B, Dai W. HEAX: An architecture for computing on encrypted data. In *Proc. the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2020, pp.1295–1309. DOI: 10.1145/3373376.3378523.

[20] Agrawal R, de Castro L, Yang G, Juvekar C, Yazicigil R, Chandrakasan A, Vaikuntanathan V, Joshi A. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *Proc. the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 25–Mar. 1, 2023, pp.882–895. DOI: 10.1109/HPCA56546.2023.10070953.

[21] Fan S, Wang Z, Xu W, Hou R, Meng D, Zhang M. TensorFHE: Achieving practical computation on encrypted data using GPGPU. In *Proc. the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 25–Mar. 1, 2023, pp.922–934. DOI: 10.1109/HPCA56546.2023.10071017.

[22] Samardzic N, Feldmann A, Krastev A, Manohar N, Genise N, Devadas S, Eldefrawy K, Peikert C, Sanchez D. CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In *Proc. the 49th Annual International Symposium on Computer Architecture*, Jun. 2022, pp.173–187. DOI: 10.1145/3470496.3527393.

[23] Kim S, Kim J, Kim M J, Jung W, Kim J, Rhu M, Ahn J H. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *Proc. the 49th Annual International Symposium on Computer Architecture*, Jun. 2022, pp.711–725. DOI: 10.1145/3470496.3527415.

[24] Kim J, Lee G, Kim S, Sohn G, Rhu M, Kim J, Ahn J H. ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *Proc. the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2022, pp.1237–1254. DOI: 10.1109/MICRO56248.2022.00086.

[25] Kim J, Kim S, Choi J, Park J, Kim D, Ahn J H. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proc. the 50th Annual International Symposium on Computer Architecture*, Jun. 2023, Article No. 18. DOI: 10.1145/3579371.3589053.

[26] Dai W, Sunar B. cuHE: A homomorphic encryption accelerator library. In *Proc. the 2nd International Conference on Cryptography and Information Security in the Balkans*, Sept. 2015, pp.169–186. DOI: 10.1007/978-3-319-29172-7_11.

[27] Putra, Prasetiyo P, Chen Y, Kim J, Kim J Y. Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping. In *Proc. the 56th IEEE/ACM International Symposium on Microarchitecture*, Oct. 28–Nov. 1, 2023, pp.1319–1331. DOI: 10.1145/3613424.3614264.

[28] Prasetiyo, Putra A, Kim J Y. Morphling: A throughput-maximized TFHE-based accelerator using transform-domain reuse. In *Proc. the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Mar. 2024, pp.249–262. DOI: 10.1109/HPCA57654.2024.00028.

[29] Nam K, Oh H, Moon H and Paek Y. Accelerating N-bit operations over TFHE on commodity CPU-FPGA. In *Proc. the 41st IEEE/ACM International Conference on Computer-Aided Design*, Oct. 30–Nov. 3, 2017.

[30] Clark L T, Vashishtha V, Harris D M, Dietrich S, Wang Z. Design flows and collateral for the ASAP7 7nm FinFET predictive process design kit. In *Proc. the 2017 IEEE International Conference on Microelectronic Systems Education (MSE)*, May 2017, pp.1–4. DOI: 10.1109/MSE.2017.7945071.

[31] Shafaei A, Wang Y, Lin X, Pedram M. FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices. In *Proc. the 2014 IEEE Computer Society Annual Symposium on VLSI*, Jul. 2014, pp.290–295. DOI: 10.1109/ISVLSI.2014.94.

[32] O'Connor M, Chatterjee N, Lee D, Wilson J, Agrawal A, Keckler S W, Dally W J. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *Proc. the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2017, pp.41–54. DOI: 10.1145/3123939.3124545.

[33] Agrawal R, de Castro L, Juvekar C, Chandrakasan A, Vaikuntanathan V, Joshi A. MAD: Memory-aware design techniques for accelerating fully homomorphic encryption. In *Proc. the 56th IEEE/ACM International Symposium on Microarchitecture*, Oct. 28–Nov. 1, 2023, pp.685–697. DOI: 10.1145/3613424.3614302.

[34] Yudha A B W, Xue J, Lou Q, Zhou, and Solihin Y. BoostCom: Towards efficient universal fully homomorphic encryption by boosting the word-wise comparisons. In *Proc. the 2024 International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2017, pp.121–132. DOI: 10.1145/3656019.3676893.

Ying-Hao Yang received his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2025. He is currently an assistant researcher with the State Key Laboratory of Processors, ICT, CAS, Beijing. His research interests include privacy-preserving computing acceleration, fully homomorphic encryption hardware acceleration, fully homomorphic processor design, and trusted execution environment.

Xi-Cheng Xu is currently pursuing his doctoral degree with the Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, Beijing. His research interests include fully homomorphic encryption acceleration, software-hardware co-acceleration, and fully homomorphic processor design.

Hang Lu received his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2015. He is currently a professor with ICT, CAS, and University of Chinese Academy of Sciences, Beijing. His research interests include fully homomorphic encryption (FHE) acceleration, FPGA accelerator design, and trusted execution environment.

Xiao-Wei Li received his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1991. In 2000, he joined ICT, CAS, as a professor. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks.

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

计算机科学技术学报（英文）

Volume 41 Number 3 2026

Bimonthly, Started in 1986 Index in: SCIE, EI, Scopus, INSPEC, DBLP, etc.
Edited by: Editorial Board of Journal of Computer Science and Technology
Editor-in-Chief: Zhi-Wei Xu; Managing Editor: Feng-Di Shu; P.O. Box 2704, Beijing 100190, P.R. China
E-mail: jcst@ict.ac.cn; Available Online: https://link.springer.com/journal/11390, https://jcst.ict.ac.cn

Sponsored by: Institute of Computing Technology, CAS & China Computer Federation
Undertaken by: Institute of Computing Technology, CAS; Supervised by: Chinese Academy of Sciences
Published by: Science Press, Beijing, China; Printed by: Beijing Baochang Color Printing Co., Ltd.
Distributed by: China—All Local Post Offices
Others—Springer Nature Customer Service Center GmbH, Germany

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY
Volume 41, Number 3, May 2026

Content