GIF-FHE: A Comprehensive Implementation and Evaluation of GPU-Accelerated FHE With Integer and Floating-Point Computing Power

Fangyu Zheng, Guang Fan, Wenxu Tang, Yixuan Song, Tian Zhou, Yuan Zhao, Jiankuo Dong, Jingqiang Lin, *Senior Member, IEEE*, Shoumeng Yan, and Jiwu Jing, *Member, IEEE*

***Abstract*—Fully Homomorphic Encryption (FHE) allows computations on encrypted data without revealing the plaintext, garnering significant interest from both academic and industrial communities. However, its broader adoption has been hindered by performance limitations. Consequently, researchers have turned to GPUs for efficient FHE implementation. Nevertheless, most have predominantly favored integer units due to their ease of use, overlooking the considerable computational potential of floating-point units in GPUs. Recognizing this untapped floating-point computational power, our article introduces GIF-FHE, an extensive exploration and implementation of FHE, leveraging GPUs' integer and floating-point instructions for FHE acceleration. We develop a comprehensive suite of low-level and middle-level FHE primitives, offering multiple implementation variants with support for three word size configurations ($64/52/32$-bit). Particularly, we make innovative use of floating-point implementations, employing a novel methodology to efficiently leverage the floating-point unit's fused multiply-add (FMA) instructions. This represents the pioneering integration of floating-point units into FHE acceleration. To bridge our highly-optimized FHE primitives with practical applications, this article also provides a high-level FHE implementation and interfaces that can be directly applied by upper-level applications such as neural network inference. Finally, we undertake a comprehensive experiment evaluation and comparison involving three types of arithmetic: FP64/INT64/INT32 with varying word size configurations and computation units. Notably, our fundamental function implementations consistently outperform counterparts on the same platform, achieving speedups ranging from $2.0\times$ to $4.2\times$. In the context of CKKS FHE schemes, our homomorphic operation implementation surpasses the state-of-the-art GPU-based solution with a speedup of up to $3.8\times$, and exceeds the performance of the widely adopted CPU-based library, SEAL, with a remarkable speedup of over $300\times$.**

***Index Terms*—FHE, GPU acceleration, floating-point arithmetic, number-theoretic transform.**

Received 16 November 2023; revised 7 November 2024; accepted 18 May 2025. Date of publication 28 May 2025; date of current version 20 June 2025. This work was supported in part by the National Cryptologic Science Fund of China under Grant 2025NCSF02005, in part by the National Natural Science Foundation of China under Grant 61902392, in part by Major Science and Technology Demonstration Project of Jiangsu Provincial Key R&D Program under Grant BE2022081, and in part by CCF-Ant R&D Fund under Grant CCF-AFSG RF20230206. Recommended for acceptance by A. Randles. *(Fangyu Zheng and Guang Fan contributed equally to this work.) (Corresponding author: Guang Fan.)*

Fangyu Zheng and Jiwu Jing are with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: zhengfangyu@ucas.ac.cn; jwjing@ucas.ac.cn).

Guang Fan, Yixuan Song, Yuan Zhao, and Shoumeng Yan are with Ant Group, Hangzhou 310023, China (e-mail: fanguang.fg@antgroup.com; songyixuan.syx@antgroup.com; caesar.zy@antgroup.com; shoumeng.ysm@antgroup.com).

Wenxu Tang, Tian Zhou, and Jingqiang Lin are with the Beijing Research Institute, University of Science and Technology of China, Beijing 100049, China (e-mail: vx.tang@qq.com; weekdayzt@gmail.com; linjq@ustc.edu.cn).

Jiankuo Dong is with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China (e-mail: djiankuo@njupt.edu.cn).

Digital Object Identifier 10.1109/TPDS.2025.3574481

I. Introduction

FULLY Homomorphic Encryption (FHE) has become an attractive "panacea" for data privacy, especially in the field of privacy-preserving computation. FHE allows performing arbitrary computation on encrypted data without the need for the secret key, hence there is no need for the knowledge of original data. A typical application of FHE is outsourcing the encrypted data to a commercial cloud environment for processing, and then decrypting the encrypted processed results, which effectively solves the problem of data confidentiality while entrusting data and its operations to a third party.

FHE dates back to 1978 [1], and achieves a breakthrough in 2009 with Gentry's blueprint [2], which is however theoretically feasible but highly impractical. Since then further efforts have been made to advance FHE to real-world usage step by step during the past decade. A series of representative FHE (or leveled FHE) schemes, such as BGV [3], BFV [4], CKKS [5] and TFHE [6] have been widely used in both industry and academia [7], [8], [9].

Despite the significant advances in FHE algorithms, substantial performance overheads remain a critical bottleneck that limits their broader deployment in real-world applications. These overheads, primarily due to intensive computation requirements, hinder the practical adoption of FHE in scenarios demanding efficient and responsive data processing. Consequently, the urgent need for FHE acceleration solutions has become increasingly prominent, drawing widespread attention from both academia and industry.

For the convenience of research and proof-of-concept, most previous FHE researches or applications heavily rely on FHE libraries running on CPU platforms, e.g., the well-known SEAL [10] and HElib [11]. These libraries focus more on functionality, usability, and compatibility across multiple platforms while sacrificing platform-specific performance optimizations.

Recent study [12] has illustrated 10^5 to 10^7 times of performance degradation for computation on encrypted data with the SEAL and HElib, compared to that on plaintext messages. In pursuit of extreme performance, many previous efforts were made to leverage the platform features to accelerate FHE schemes. Boamer et al. [13] put forward Intel HEXL to accelerate NTT and modular multiplication with Intel AVX512-IFMA instruction, achieving a $7.2\times$ single-threaded speed up. However, limited by the performance upper bound of the CPU platforms, these implementations cannot yet meet the requirements of large-scale data processing.

A. Opportunities and Challenges for GPU-Based FHE

Fortunately, the rapid advancements in graphics processing units (GPUs) bring an opportunity to address the performance challenges associated with Fully Homomorphic Encryption (FHE). NVIDIA's introduction of the Single Instruction, Multiple Threads (SIMT) execution model in 2006 revolutionized GPU parallel computing. SIMT leverages thread-level parallelism, allowing multiple independent threads to execute concurrently using a single instruction. This feature holds great promise for accelerating FHE, which involves intensive arithmetic operations. Prior research endeavors have made significant efforts to harness the computational power and memory bandwidth of GPUs. By transplanting and parallelizing conventional CPU-based implementations, GPU-based implementations have achieved performance improvements ranging from tens to hundreds of times faster.

Previous studies [14], [15], [16], [17], [18], [19] have leveraged GPU to accelerate the fundamental operations (e.g., CRT and NTT) of FHE. And some previous works [20], [21], [22] are dedicated to exploring efficient NTT implementations on GPU for FHE. Based on the GPU FHE implementation, Al Badawi et al. [23] present a text classification solution, PrivFT, using FHE to preserve the privacy of content.

Nonetheless, the choice of underlying computational units and implementation algorithms significantly influences the performance of FHE implementations. Many prior works have followed CPU-based implementation patterns, even though CPUs and GPUs handle distinct types of workloads, leading to a lack of effective utilization of GPU-specific advantages.

For instance, consider the underlying instruction set. High-definition 3D graphics processing and deep learning applications demand high-speed floating-point processing capabilities. GPUs inherently excel in floating-point computing power. Over the past decade, NVIDIA GPUs have witnessed more than a ten-fold increase in floating-point computing power, growing from 1.345 Tera Floating-point Operations Per Second (TFLOPS) in the Fermi architecture to 40 TFLOPS in the Ampere architecture. Some GPU generations, such as Maxwell and Pascal, lack fully functional integer units. Recognizing the significant potential of floating-point computing power, previous studies [24], [25], [26] reported up to a $3\times$ speedup for RSA and ECC when utilizing the floating-point computing capability of GPUs, compared to integer-based implementations. Previous works also explore the use of floating-point computing power in other domains [27], [28], [29], [30]. However, no prior research has explored the application of GPUs' floating-point computing power in FHE implementations.

Furthermore, there is room for further exploration in the implementation of CRT and NTT algorithms used in FHE. Adaptation to different GPUs, underlying arithmetics, and implementation methods requires empirical validation. Moreover, these algorithms, particularly the NTT algorithm, offer opportunities for optimization.

B. Contributions and Paper Organization

This paper aims to provide a comprehensive implementation (and evaluation) of FHE schemes with fine-grained optimization across a range of computing units in **G**PUs, including **I**NT32 units and **F**P64 units, in pursuit of a faster **FHE** implementation (**GIF-FHE**). Our contributions can be summarized as follows:

- First, we thoroughly explore the computing resources inherent in GPUs by effectively utilizing the INT32 compute units and innovatively incorporating FP64 compute units in FHE acceleration. Subsequently, we fully leverage the computational capabilities of GPUs for three word size configurations ($64/52/32$-bit) and provide three comprehensive sets of underlying arithmetic: INT64/FP64/INT32 arithmetic. This offers multiple choices for cryptographic primitives implementations in higher-level constructions. Additionally, we design targeted experiments to conduct extensive evaluations of the three sets of underlying arithmetic. We conduct a detailed analysis of the strengths and weaknesses of these arithmetic operations in terms of generality and performance within homomorphic operators, providing valuable insights for future research.
- Second, we extensively exploit the multi-level memory architecture and various thread synchronization methods of GPUs and building upon the underlying arithmetic operations, we provide a complete set of polynomial operations for FHE schemes based on the Ring Learning With Errors (RLWE) problem, including NTT/INTT and CRT/ICRT. We meticulously employ multiple optimization techniques and compare implementations using different strategies. We benchmark our work against state-of-the-art GPU-based approaches, and our optimized implementation achieves a performance improvement of $2.0 \sim 4.2\times$ on the same platform.
- Third, as a case study, we employ the CKKS algorithm and utilize device-level kernel enhancement and memory management techniques to organically arrange and develop polynomial arithmetic kernels. This enables us to efficiently implement homomorphic operations and further achieve privacy-preserving inference applications. We conduct extensive testing and comparisons of key-switching, homomorphic multiplication, and rotation operations, revealing that our implementations achieve a maximum improvement of up to $3.8\times$ compared to the top-performing CUDA Core-based implementations, and over $300\times$ compared to the widely adopted CPU-based library. Utilizing FP64 arithmetic in the key-switching operations, we have

observed that on the P100 GPU, the performance can reach up to $2.9\times$ that of the INT64 arithmetic implementation. Additionally, our privacy-preserving inference on a single image showcases a remarkable $49.8\times$ improvement compared to the state-of-the-art CPU implementation.

The rest of our paper is organized as follows. Section II presents background material. Section III gives an overall architecture of the GIF-FHE. Sections IV, V and VI detail the implementation of underlying arithmetic, polynomial arithmetic and upper homomorphic operations, respectively. Section VII analyses the performance of proposed algorithm and compares it with previous works. Section VIII concludes the paper.

II. Background

This section serves as a cornerstone for grasping the core concepts essential to understanding the paper. It commences by presenting the notations used throughout the paper, followed by an introduction to the CKKS scheme and the fundamental algorithms required for efficient implementation of FHE. Lastly, a concise overview is provided regarding the floating-point numbers and GPUs' FMA instruction.

A. Notation

$\mathcal{R}_q$ is the polynomial ring $\mathbb{Z}_q[X]/\{X^N+1\}$, where N is a power of 2. For a polynomial $a(X)=\sum_{i=0}^{N-1} a_i \cdot X^i$, we define $\mathbf{a}=(a_0, a_1, \ldots, a_{N-1})$. The symbol $L(\ell)$ indicates the maximum (current) level of a ciphertext, the number of multiplications one can perform on the ciphertext without decryption. r represents the number of prime numbers to represent large integers in RNS domain. β refers to the word size, indicating the number of bits processed by a multiplication instruction (or simulated multiplication instruction).

B. Prevailing FHE Schemes

Currently, the prevailing FHE Schemes includes CKKS [5], BGV [3], BFV [4]. This section will take CKKS as an example to demonstrate the basic functions of FHE schemes.

CKKS supports homomorphic operations on real numbers, rendering it conducive to data science applications. In CKKS, a message is a vector of $N/2$ complex numbers. CKKS encodes it into a polynomial $m(X) \in \mathcal{R}_{Q_L}$, and then encrypts the plaintext $m(X)$ as a level L ciphertext $\mathrm{ct}=(a(X), b(X))$ in $R^2_{Q_L}$, a polynomial pair, where $Q_L(Q_\ell)$ represents the ciphertext modulus at level $L(\ell)$.

The key-switching technique is widely used in FHE schemes. In 2020, Han et al. [31] proposed a generalized key-switching technique, along with the introduction of a new parameter, `dnum`, intended to strike a balance between the complexity of key-switching and the scheme's ability to accommodate a larger number of levels. In this paper, we adopt a straightforward approach by setting $\mathtt{dnum}=L+1$, which allows for a greater number of multiplication layers. Given an evaluation key $\mathbf{evk}$ and a polynomial $[d(X)]_{Q_\ell}$, the procedure $\mathtt{KeySwitch}(d(X), \mathbf{evk})$ decomposes the input polynomial $[d(X)]_{Q_\ell}$ into $[d(X)]_{q_j}$, where $j \in [0, \ell+1)$, raises its moduli to obtain $[\tilde{d}(X)]_{Q_{\ell+1}}$, then computes $(a^{(i)}(X), b^{(i)}(X) = \Sigma_{j=0}^{\ell} \tilde{d}_j^{(i)} \odot \mathbf{evk}_j^{(i)}$ for $i \in [0, \ell+1]$ and finally reduces the moduli to obtain $[(a(X), b(X)]_{Q_\ell}$.

Below, we present the most frequently used FHE evaluation functions in CKKS. For more details, refer to [5], [31], [32]:

- *Homomorphic Addition (*`HADD`*):* Given two ciphertexts $\mathrm{ct}_0=(a_0(X), b_0(X))$ and $\mathrm{ct}_1=(a_1(X), b_1(X))$ in $R^2_{Q_\ell}$, $\mathtt{HADD}(\mathrm{ct}_0, \mathrm{ct}_1) := [a_0(X)+a_1(X), b_0(X)+b_1(X)]_{Q_\ell}$.
- *Homomorphic Multiplication (*`HMULT`*):* Given two ciphertexts $\mathrm{ct}_0=(a_0(X), b_0(X))$ and $\mathrm{ct}_1=(a_1(X), b_1(X))$ in $R^2_{Q_\ell}$ and an evaluation key $\mathbf{evk}$, $\mathtt{HMULT}(\mathrm{ct}_0, \mathrm{ct}_1) := [(a_0(X)\cdot a_1(X), a_0(X)\cdot b_1(X)+a_1(X)\cdot b_0(X)) + \mathtt{KeySwitch}(b_0(X)\cdot b_1(X), \mathbf{evk})]_{Q_\ell}$.
- *Plaintext-ciphertext Multiplication (*`CMULT`*):* Given a plaintext $\mathrm{m}=(a_0(X)) \in R$ and a ciphertext $\mathrm{ct}=(a_1(X), b_1(X))$ in $R^2_{Q_\ell}$, $\mathtt{CMULT}(\mathrm{m}, \mathrm{ct}) := [(a_0(X)\cdot a_1(X), a_0(X)\cdot b_1(X))]_{Q_\ell}$.
- *Homomorphic Rotation (*`HROTATE`*):* Given a ciphertext $\mathrm{ct}=(a_0(X), b_0(X))$ in $R^2_{Q_\ell}$, a rotation index rn and an evaluation key $\mathbf{evk}$, `HROTATE` rotates the message of ct according to the rn, which is defined as $\mathtt{HROTATE}(\mathrm{ct}, rn, \mathbf{evk}) := [(0, \mathrm{rotate}(b_0(X), rn) + \mathtt{KeySwitch}(\mathrm{rotate}(a_0(X), \mathbf{evk})]_{Q_\ell}$.
- *Rescale:* Given a ciphertext $\mathrm{ct}=(a(X), b(X))$ in $R^2_{Q_\ell}$, `RESCALE` performs rescaling separately to $a(X)$ and $b(X)$, yielding the result $ct'=(a'(X), b'(X)) \in R^2_{Q_{\ell-1}}$.

C. Typical FHE Workloads

The polynomial arithmetic, especially the polynomial multiplications, poses as the primary computational bottleneck of main word-wise FHE schemes, like BGV, BFV, and CKKS. The Chinese Remainder Theorem (CRT) and Number Theoretic Transform (NTT) are widely used in FHE implementations to reduce the complexity of polynomial arithmetic.

1) Chinese Remainder Theorem (CRT): CRT is used to transform the polynomial from raw multi-precision representation to CRT representation by decomposing large coefficients into a set of single-word coefficients.

To exploit CRT, r co-prime moduli q_i for $i \in \{0 \le i < r\}$ need to be selected, where $Q=\prod_{i=0}^{r-1} q_i$ and each modulus q_i is a prime number smaller than word size β. Then, a integer a for $a<Q$ can be represented by the set of remainders $\{a_0, a_1, \ldots, a_{r-1}\}$, where $a_i = a \pmod{q_i}$. Therefore, the arithmetic over $\mathbb{Z}_Q$ can be transformed to arithmetic over the finite field $\mathbb{Z}_{q_i}$ for $i \in \{0 \le i < r\}$, e.g., a multi-word multiplication is converted to r single-word modular multiplications. It not only reduces the computational complexity from $O(m^2)$ to $O(r)$ for $m=\lceil \log Q/\log \beta \rceil$ ($r \ll m^2$), but also can be processed in parallel.

The reconstruction of big integer a is called ICRT, which can be carried out via (1).

$$a=\sum_{i=0}^{r-1} \frac{Q}{q_i} \cdot \left(\left(\frac{Q}{q_i} \right)^{-1} \cdot a_i \bmod q_i \right) \bmod Q \tag{1}$$

2) Number Theoretic Transform (NTT): NTT is a specialized form of the Discrete Fourier Transform (DFT) for a finite field of integers. Using NTT, one can transform the polynomial from coefficient representation to NTT representation (point-value representation). Then the polynomial multiplication can be converted into element-wise multiplication operations of 2 vectors in NTT form. The computational complexity of the multiplication between two polynomials in the polynomial ring is reduced from $O(N^2)$ to $O(N \log N)$.

The NTT algorithm for the polynomial of degree N, which is also called N-point NTT, computes the following: $A_k = \sum_{j=0}^{N-1} a_j \omega_N^{jk} \bmod q_i$ for $k \in \{0 \leq k < N\}$, ω_N is the primitive Nth root of unity of NTT for $\mathbb{Z}_{q_i}$ ($\omega_N^{jk} \bmod q_i$ is called twiddle factors), a_j is an input polynomial coefficient indexed by j, and A_k is an output polynomial coefficient indexed by k.

Generally, when using NTT, it is required to double the input polynomial with zero-padding to obtain the result of $2N-1$ degree and modulo the polynomial modulus after element-wise multiplication. For polynomial ring $\mathbb{Z}_{q_i}[X]/\{X^N+1\}$, which is used in most FHE schemes, a technique called negacyclic convolution [33] is involved to avoid those operations. The process is shown in (2), in which $\mathbf{\Omega^{-1}} = (1, \omega_{2\tilde{}N}^{-1}, \omega_{2\tilde{}N}^{-2}, \ldots, \omega_{2\tilde{}N}^{1-N})$.

$$\mathbf{c} = \mathbf{\Omega^{-1}} \odot \mathrm{INTT}_N\left(\mathrm{NTT}_N\left(\bar{\mathbf{a}}\right) \cdot \mathrm{NTT}_N\left(\bar{\mathbf{b}}\right)\right) \qquad (2)$$

$\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$ are the vectors of the coefficients of $a(\omega_{2N}X)$, $b(\omega_{2N}X)$ respectively. Consider $\mathrm{NTT}(\bar{\mathbf{a}}) = \mathbf{A}$, $A_k = \sum_{j=0}^{N-1}(a_j\omega_{2\tilde{}N}^{j})\omega_N^{jk}$. To reduce the number of computations, it is usually simplified to $A_k = \sum_{j=0}^{N-1} a_j \omega_{2\tilde{}N}^{j(2k+1)}$. The transform of $\mathrm{NTT}(\bar{\mathbf{a}})$ is also called Discrete Weighted Transform (DWT) [34]. Similarly, $\mathrm{INTT}(\mathbf{A} \cdot \mathbf{\Omega^{-1}})$ is the inverse of DWT (IDWT).

Compared with traditional iterative NTT, four-step NTT [35] is more suitable for parallel computing. In four-step NTT algorithm, the length N of polynomial is decomposed into (N_1, N_2) for $N = N_1 \times N_2$. In this way, $A_{(k_2,k_1)}$ in $\mathbf{A} = NTT(\mathbf{a})$ can be computed as (3). In which, $j = j_1 + j_2 N_1$, $k = k_2 + k_1 N_2$.

$$A_{(k_2,k_1)} = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} a_{(j_1,j_2)} \omega_{N_2}^{j_2 k_2} \omega_{N_1 N_2}^{j_1 k_2} \omega_{N_1}^{j_1 k_1} \bmod q_i \qquad (3)$$

The above equation reveals the new four steps of the algorithm: 1) proceeding N_1 groups of N_2-point NTT; 2) transposing the matrix; 3) multiplying twiddle factors; and 4) proceeding N_2 groups of N_1-point NTT.

3) Limitations of Existing GPU-Based Solutions: Despite significant advancements in GPU-based FHE implementations, several limitations persist across different levels of the architecture, affecting performance and efficiency.

Underlying arithmetic level: Existing GPU-based solutions [14], [15], [16], [17], [18], [19] often rely on limited arithmetic strategies that do not fully utilize the GPU's computational potential, which directly impacts the efficiency of foundational workloads like CRT and NTT. Many implementations focus on single arithmetic types (e.g., INT32), which restricts flexibility and efficiency, especially when handling the varied arithmetic demands of polynomial transformations. This single-method approach fails to explore the benefits of leveraging different arithmetic units, such as FP64, which can improve performance in certain scenarios by exploiting the GPU's floating-point capabilities. Additionally, the complexity of multi-word arithmetic and the overhead of modular operations in finite fields, essential for CRT processes, remain challenges that are inadequately addressed.

Polynomial arithmetic level: The main workloads in FHE, such as CRT and NTT for polynomial arithmetic, experience inefficiencies due to suboptimal utilization of memory and computational resources. Current implementations frequently encounter challenges with memory access patterns and synchronization issues, leading to increased latency and reduced throughput, which are critical in CRT and NTT tasks. For instance, operations like NTT/INTT require efficient memory hierarchy utilization, which is not fully exploited in many solutions, resulting in unnecessary global memory accesses and synchronization delays. This inefficiency limits the potential speedup and scalability that GPUs can offer for these complex polynomial operations.

Homomorphic operations level: On a device level, the implementation of homomorphic operations like `HMULT` and `HROTATE`, which depend on optimized polynomial arithmetic, often faces challenges due to limitations in kernel deployment and memory management strategies. Many solutions use kernels that handle single polynomials and single polynomial operations for complex homomorphic tasks, and the lack of kernel enhancement techniques results in poor optimization of resource utilization and extended execution times Furthermore, memory management in existing solutions often overlooks advanced allocation strategies, leading to high latency and inefficient bandwidth usage, particularly when dealing with large-scale data inherent to FHE operations.

Overall, these limitations highlight the need for a more integrated and optimized approach that addresses arithmetic efficiency, workload management, and device-level execution to enhance the performance of GPU-based FHE solutions.

D. Floating-Point Numbers and the FMA Operation

The 32-bit and 64-bit basic binary floating-point formats defined in IEEE 754 [36] correspond to the C language data types `float` and `double`. This guarantees consistent computations across platforms and convenient exchange of data. The double precision floating point (FP64) value used in this work has a 1-bit sign, an 11-bit exponent, and a 53-bit significand. The 53-bit significand includes an implicit integer bit of value 1 and an explicit 52-bit fraction part.

The fused multiply-add operation (FMA), also included in IEEE 754 [36], computes $(X \times Y + Z)$ with only one rounding step. This mechanism makes FMA more accurate than performing separate multiply and add operations with two rounding steps. NVIDIA GPUs with CUDA architecture comply with the IEEE 754 standard for binary floating-point arithmetic with acceptable deviations.

There are five rounding modes defined by IEEE 754 including *round-to-nearest* (ties to even, ties away from zero), *round towards positive*, *round towards negative*, and *round towards zero*. Among them, *round towards zero* (`rz`) only retains the effective digit precision and directly discards the extra digits.

III. GIF-FHE Design

This section gives an overview and roadmap of GIF-FHE.

A. A Multi-Layer Design of GIF-FHE

FHE encompasses various complex algorithms and operations. A multi-layer architecture is particularly suited for managing such intricate systems. Previous work leverages multi-layer parallel framework techniques on GPUs in other fields [37].

In this study, we employ a multi-layer framework on GPUs for FHE acceleration, which offers several advantages. First, this design allows for specific performance optimizations at each layer, enabling the effective application of diverse optimization strategies. Second, it breaks down complex operations into smaller modules, allowing for independent replacement or upgrading of different layers. This capability enables the system to be adjusted and optimized according to diverse usage needs and quickly adapt to new technologies or algorithm changes, thereby improving the system's flexibility and scalability.

B. Roadmap of GIF-FHE

GIF-FHE introduces three distinct layers: the polynomial arithmetic layer, the underlying arithmetic layer, and the homomorphic operation layer.

The underlying arithmetic layer, as detailed in Section IV, is meticulously designed to operate on individual coefficient levels with remarkable efficiency. By leveraging GPU instruction sets, this layer achieves high-performance computing through primitive-level optimization.

The polynomial arithmetic layer, explored in Section V, concentrates on operations within the entire polynomial structure. Here, we employ kernel-level optimization to significantly enhance performance by using advanced thread organization and inter-thread synchronization to capitalize on parallel processing power. By effectively coordinating the workload across various threads, it significantly boosts efficiency in handling large-scale data.

The homomorphic operation layer, described in Section VI, is tailored for supporting encryption schemes based on the RLWE problem. Our implementation focuses on the CKKS scheme, utilizing both the polynomial and underlying arithmetic layers to perform efficient homomorphic operations. This layer benefits from device-level optimization, enabling seamless homomorphic operations that fully harness the potential of the hardware. Moreover, we demonstrate privacy-preserving inference using the CKKS scheme, highlighting its practical applications in maintaining data privacy while delivering robust computational power.

TABLE I
Throughput (Number of Operations Per Clock Cycle Per SM) of Native Integer and Floating-Point Arithmetic Instructions on NVIDIA P100, A100, and H100 GPUs

Throughput/SM/CLK	FP32 FMA	FP64 FMA	INT32 Mul
P100	64	32	< 32 *
A100	64	32	64
H100	128	64	64

*: It takes multiple instructions to complete one 32-bit integer multiplication.

IV. Underlying Arithmetic

The various computing units and rich instruction set of GPUs support their abundant computing capabilities. This section aims to fully utilize the computing resources of GPUs and propose efficient FHE underlying primitives implementation. This section first analyzes the feasibility of implementing FHE mathematical primitives using different computing units, then introduces three sets of underlying arithmetic using different instructions, as well as the implementation of finite field arithmetic and multi-word arithmetic.

A. GPU Native Instruction Set Selection

There are multiple computing resources in modern GPUs, including separate arithmetic logic units (ALU) that support integer and floating-point calculations. Influenced by applications such as graphics processing and machine learning, GPU architectures are typically designed with a greater emphasis on floating-point computational capabilities. As a result, compared to integer arithmetic units, GPU floating-point computation units not only have a wider variety but also often have higher throughput.

In GPUs designed for high-performance computing and data centers such as H100 and A100, there are two types of computational cores for floating-point calculations, FP32 and FP64 cores, and one type for integer calculations, INT32 cores. In some cases where 64-bit integer calculations are required, NVIDIA GPUs simulate the implementation of it using 32-bit instructions. The instruction throughput of the NVIDIA H100, A100, and P100 GPUs is shown in Table I.

In the latest H100 GPU, the number of FP32 cores has doubled compared to the INT32 cores, resulting in FP32 calculations achieving twice the throughput of INT32 calculations. However, in the P100 GPU, which lacks fully functional integer cores, it takes multiple instructions to complete one 32-bit integer multiplication, resulting in approximately half the throughput of FP32 operations for INT32 multiplication calculations.

However, currently, the floating-point unit of GPUs has not been considered in FHE acceleration. This is largely because the use of floating-point units for integer calculations can easily lead to loss of precision. Most operations in cryptographic computations require accurate calculation results. For such computations, the utilization of FMA instructions and careful treatment are required.

With noticing that in the GPU, there are two types of floating-point units, FP32 units and FP64 units, with significant bits of 24 and 53, respectively. In FHE implementations, CRT and NTT algorithms usually require a large modulus, typically 30 bits or more. The use of FP32 units can restrict the choice of modulus size, and the proportion of significant bits in FP32 numbers is relatively low, resulting in low computational efficiency. In addition to integer units and floating point units, the latest GPUs also feature tensor cores. While some studies have explored using tensor cores for cryptographic computations such as lattice-based cryptography [38], tensor cores primarily support low word-size operations such as INT8 and FP16, making them inefficient for FHE calculations, which require larger moduli.

Based on the aforementioned considerations, we opted to build FHE implementation based on INT32 and FP64 units, respectively.

B. Three Sets of Underlying Arithmetic

The INT32 unit is capable of handling both 32-bit and 64-bit integer computations. Meanwhile, the FP64 unit, through specific adaptations, accommodates 52-bit integer computations. For clarity, we will refer to implementations with distinct word sizes as INT32 arithmetic, INT64 arithmetic, and FP64 arithmetic, respectively.

Each of these underlying arithmetics exhibits distinct characteristics. INT64 arithmetic accommodates larger finite field moduli, INT32 arithmetic exhibits higher computation speed, and FP64 arithmetic enables computations using the floating-point unit, harnessing the floating-point computational power of the GPU.

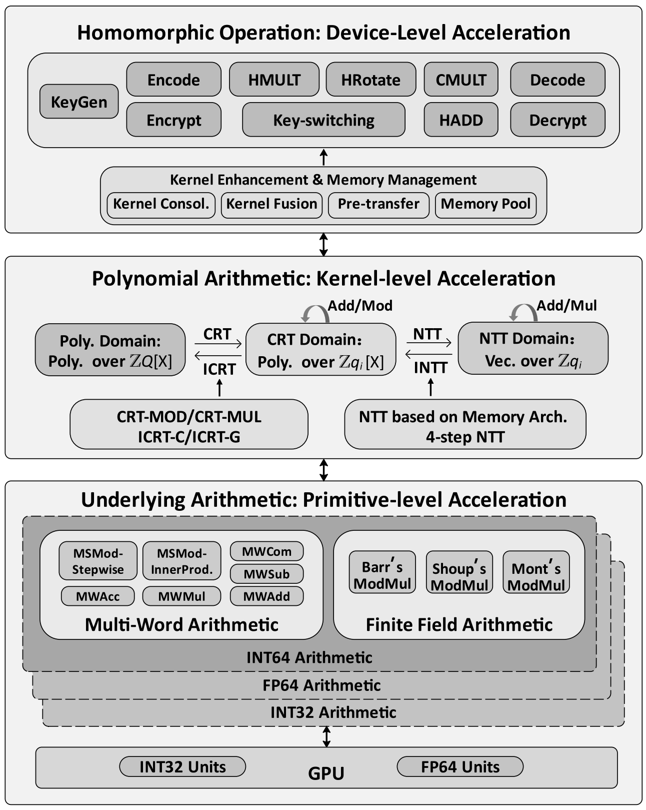
Previous studies [14], [15], [39] used 32-bit word size, while other studies [16], [20], [40] chose 64-bit word size. These studies exclusively employed a single arithmetic, without evaluating the merits of different arithmetics within the same algorithm. Besides, there is no research to demonstrate an FHE implementation using FP64 arithmetic.

Thus, we implement all three underlying arithmetic types, trying to explore the best routine for FHE implementations to achieve faster performance under varying device and parameter configurations.

The overall design is shown as Fig. 1. The underlying arithmetic includes finite field arithmetic and multi-word arithmetic, providing basic operators for the upper polynomial arithmetic. INT32 and INT64 arithmetic are based on INT32 units of GPUs and the FP64 arithmetic is based on FP64 units. Next, we will go through these three sets of underlying arithmetic.

INT32 Arithmetic: INT32 arithmetic is based on the INT32 cores in GPUs, supporting a word size of 32 bits. GPUs feature native 32-bit integer instructions, with the CUDA compiler converting operations on INT32 data types into corresponding instructions. In critical functions, we use the CUDA parallel thread execution (PTX) instruction set, which includes multiply-add instructions (`mad`) and instructions related to carry operations (`addc`, `subc`, and `madc`) to improve the efficiency of the relevant operations.

INT64 Arithmetic: There are no 64-bit integer arithmetic units in NVIDIA GPUs. While the compiler provides support for computations involving 64-bit integers, the underlying calculation is still performed using INT32 cores. An INT64 multiplication is substituted by four INT32 multiplication instructions and six addition instructions in total. INT64 arithmetic supports a word size of up to 64 bits. To optimize critical functions involving multiply-add and carry operations, we still use PTX assembly instructions.



Homomorphic Operation: Device-Level Acceleration
KeyGen
Encode
HMULT
HRotate
CMULT
Decode
Encrypt
Key-switching
HADD
Decrypt
Kernel Enhancement & Memory Management
Kernel Consol.
Kernel Fusion
Pre-transfer
Memory Pool
Polynomial Arithmetic: Kernel-level Acceleration
Add/Mod
Add/Mul
Poly. Domain: Poly. over $\mathbb{Z}Q[X]$
CRT
ICRT
CRT Domain: Poly. over $\mathbb{Z}q_i[X]$
NTT
INTT
NTT Domain: Vec. over $\mathbb{Z}q_i$
CRT-MOD/CRT-MUL ICRT-C/ICRT-G
NTT based on Memory Arch. 4-step NTT
Underlying Arithmetic: Primitive-level Acceleration
MSMod-Stepwise
MSMod-InnerProd.
MWCom
MWSub
MWAcc
MWMul
MWAdd
Multi-Word Arithmetic
Barr's ModMul
Shoup's ModMul
Mont's ModMul
Finite Field Arithmetic
INT64 Arithmetic
FP64 Arithmetic
INT32 Arithmetic
INT32 Units
GPU
FP64 Units

Fig. 1. Roadmap of GIF-FHE.

FP64 Arithmetic: FP64 arithmetic combines the advantages of floating-point and integer computing power in GPUs, resulting in optimal performance for both multiplication and addition operations.

To perform multiplications, we utilize the FP64 instruction, which is twice as fast as INT64 multiplication in some GPUs and performs comparably in others. We draw inspiration from the previous work's utilization of the `fma` instruction [25] and propose a computation method for single-word multiplication (SWMul) in FP64 arithmetic, as shown in Algorithm 1. By exploiting all fraction part of FP64 numbers, the word size of FP64 Arithmetic reaches 52-bit.

Specifically, we utilize two `fma` instructions to calculate the high 52-bit and the low 52-bit of the product of SWMul, respectively. In order to fix the most significant bit of the `fma` results, we set the third operand of `fma` instruction as an *anchor* value. In the line 3, we set the *anchor* to 2^{104} to get f_hi, the high 52-bit product with *anchor*. In the result, the anchor just occupies the implicit bit. Note that the rounding mode of `fma` instruction is set to *round towards zero*, so that the f_hi will not be affected by the low 52-bit product. Next, we subtract the high 52-bit from the product, and fix the most significant bit of f_lo by setting the *anchor* value to 2^{52}, which is equivalent to setting the

Algorithm 1: SWMul in FP64 Arithmetic.

```
Require: single-word integer $a$ and $b$.
Ensure: $(h, l) = a \times b$.
1: $c_1 = 2^{104}, c_2 = 2^{104} + 2^{52}, mask = 2^{53} - 1$
2: convert $a$ and $b$ to FP64 type
3: $f\_hi = \texttt{fma.f64.rz}(a, b, c_1)$
4: $f\_lo = \texttt{fma.f64.rz}(a, b, c_2 - f\_hi)$
5: $\mathbf{mov.d64}\ \ h, f\_hi$
6: $\mathbf{mov.d64}\ \ l, f\_lo$
7: $h = h \,\&\, mask$
8: $l = l \,\&\, mask$
```

third operand of the fma instruction to $(2^{52} - (f\_hi - 2^{104}))$. In this way, we get the low 52-bit of the product with *anchor* in line 4. Due to the introduction to the *anchor*, the implicit leading bit is sacrificed and thus only 52 bits are available in all 53-bit significand of FP64 numbers.

Note that in FP64 arithmetic, numbers are stored as 64-bit integers. We will explain the benefits of this later. In this case, the data needs type conversion before and after multiplications. The overhead of converting the input multiplicands to floating-point numbers is negligible, but converting floating-point numbers back to integer numbers is a time-consuming operation. Due to the introduction to the *anchor*, we can easily extract the INT64 type results by exploiting the native mov and and bit-wise instructions. And the *anchor* itself is implicit and does not require further operations.

In FP64 arithmetic, the integer computing power is also exploited to handle additions and subtractions for two reasons. First, there is no FP64 instruction to efficiently resolve carry operations. Second, the instruction throughput of integer addition is relatively high in almost all generations of GPUs, unlike the integer multiplication instructions. In this case, coefficients are stored in 64-bit integer arrays. When performing addition operations, the word size is temporarily extended to 64-bit as a redundant representation. The extra 12 bits are used to prevent overflow without carry operations, which can greatly improve the accumulation efficiency.

C. Multi-Word Arithmetic

In homomorphic encryption schemes using residue number system (RNS) representation, multi-word arithmetic is mainly used for CRT and its inverse, where multi-precision representation is used for multi-word coefficients.

Multi-word multiplication (MWMult), accumulation (MWAcc), multiplication-and-accumulation (MWMAC), and subtraction (MWSub) are frequently utilized operations. In ICRT algorithm, MWAcc and MWMAC are employed to calculate the reconstruction of large integers. The process involves intricate operations, particularly in dealing with repetitive carry operations. In INT32 or INT64 implementation, we adopt addc instructions to avoid overflow. Most of addc instructions just add zero and the carry-in flag to the summation, which is hereinafter referred to as AddcZero instructions.

MWMAC is more complicated than MWAcc. In a common MWMAC operation, a t-word integer a is multiplied by a single-word integer b and the result is accumulated to an integer array s, which does not exceed l in length. In total, $(4l - 4t + 1)$ and $(2l - 2t + 1)$ AddcZero instructions are included in one MWMAC in INT64/INT32 arithmetic respectively. The number of instructions will not reduce even if t decreases.

In FP64 implementation, each sample of a coefficient is up to 52-bit and stored as a 64-bit integer variable. The remaining 12-bit can be used to avoid overflow during accumulation and the AddcZero instructions are no longer required. So the MWMAC in FP64 arithmetic is much simpler than that in INT64 arithmetic. After the complete accumulation, we use bit operation to add the first 12-bit of each sample into the more significant sample to restore the word size to 52-bit.

Besides, we also implement multi-word subtraction (MWSub), multi-word comparison (MWCom) and multi-word integer modulo single-word integer operation (MSMod). For MWSub, we use subc instructions to improve performance in integer arithmetic. In FP64 arithmetic, an extra carry flag is introduced and the subtraction with borrow is implemented. For MWCom, the comparison is carried out from the most significant sample to the least significant sample until the result is obtained. The MSMod is the dominant computing load of CRT algorithm and significantly impacts the performance of polynomial CRT operations, which will be discussed in detail in Section V-B1.

D. Finite Field Arithmetic

In the implementation of FHE schemes, most operations are calculated in finite fields, making modular arithmetic a crucial operation for the overall performance. Finite field arithmetic includes modular addition (ModAdd), subtraction (ModSub), and multiplication (ModMul) operations.

The ModMul is the most time-consuming operation in finite field arithmetic. The inherent slowness of the native modulo operator (%) within GPU kernels poses challenges for the efficient implementation of modular arithmetic. The use of fast modular multiplication algorithms presents a solution to alleviate the computational overhead associated with ModMul. For instance, in scenarios where it is not possible to precompute either of the two multiplicands, the Barrett reduction algorithm [41] is employed.

To optimize cases where one of the multiplicands can be precomputed, we also implement the more concise Shoup's ModMul [42] and Montgomery's ModMul [43]. In the case of these two algorithms, Montgomery's ModMul entails one full multiplication (FulMul), one high-word multiplication (HiMul), and one low-word multiplication (LoMul). On the other hand, Shoup's ModMul necessitates one HiMul and two LoMuls.

It is worth noting that the computational cost of HiMul and LoMul differs among the three underlying arithmetics. In GPUs, a single instruction can accomplish 32-bit integer multiplication, whereas for INT64 arithmetic, the number of instructions required for LoMul is relatively fewer compared to FulMul and HiMul. While, in FP64 arithmetic, HiMul incurs the fewest number of instructions. Hence, in the context of INT64 arithmetic, Shoup's ModMul can be achieved with a lower number of instructions compared to Montgomery's ModMul. However, Montgomery's ModMul does not necessitate the use of the original multiplier w as input, which can provide an advantage in memory-intensive programs. Given the precomputability of rotation factors, these two algorithms are commonly employed in the NTT operation. In Section VII-B3, we conducted experimental comparisons to assess the performance of these algorithms within the context of NTT.

In addition, we employ lazy reduction to simplify computations for modular arithmetic. When selecting the modulus, we reserve 3 bits, allowing temporary values to inflate up to 2^3 times the modulus. This strategy effectively minimizes the requirement for conditional subtractions in modular multiplication, addition, and subtraction operations. As for the accumulation of modular multiplication results, we initially compute the accumulation of products and subsequently employ the Barrett reduction technique for modular reduction.

V. Polynomial Arithmetic

In homomorphic encryption implementations, polynomials are commonly represented in three different domains: polynomial domain, CRT domain, and NTT domain, as shown in Fig. 1. In order to improve the efficiency of various types of operations, polynomials need to be frequently transformed across these three domains. The main workloads of polynomial arithmetic include inter-domain conversions, polynomial addition in CRT/NTT domain, modulo conversion in CRT domain and polynomial multiplication in NTT domain. Among them, most arithmetic operations, including CRT/ICRT, demonstrate high parallelism, while NTT/INTT exhibits a strong data correlation. In this section, we will introduce NTT and other polynomial arithmetics separately.

A. NTT Optimization With Multiple Levels of GPU Memory Hierarchy

Due to the inability of polynomials in NTT form to perform modulus switching, and the presence of frequent modulus conversion operations in homomorphic encryption schemes, the NTT form of the polynomial cannot be maintained consistently. This necessitates the use of multiple NTT and INTT operations in various homomorphic operations, such as homomorphic multiplication (`HMULT`). To attain the desired $O(N \log N)$ computational complexity, the NTT algorithm incorporates several iterative rounds, with numerous memory access operations occurring between each round. Additionally, inter-thread data synchronization is required between those rounds. All the above pose significant challenges for parallel algorithm design.

As a result, the primary design principle of NTT/INTT in GIF-FHE is to enhance parallelism effectively, making full use of the ample and high-speed register resources, along with the relatively constrained in size but intra-block-synchronized shared memory. This approach aims to minimize unnecessary slow global memory access. Additionally, we employ memory-efficient algorithms like Montgomery's ModMul and the 4-step NTT, which are all characterized by their friendly memory access patterns, in order to further reduce memory access as much as possible.

1) Memory Usage and Thread Synchronization: The limited on-chip memory resources (registers and shared memory) cannot accommodate the entire polynomial and lack data synchronization capabilities between blocks. Thus, the off-chip global memory is essential due to its ample size, but its slow read and write speeds often lead to computational core delays caused by numerous memory transactions. Moreover, using device memory for data synchronization incurs significant overhead, requiring the termination of the current kernel and the initiation of a new one for subsequent tasks. To tackle these issues, this section leverages hierarchical memory properties and introduces an NTT scheme based on GPU memory architecture.

Register-Level Optimizations: It's worth mentioning that GPUs have relatively ample register resources, with up to 255 32-bit registers per thread. The fundamental idea behind our register-level NTT optimizations is to judiciously reduce parallelism, consolidate workloads, and, by doing so, increase register utilization. This, in turn, allows for a higher number of operations to be executed per iteration, ultimately reducing the overall number of iterations required.

Fig. 2 demonstrate an example of this insight. In a naive NTT implementation for a 2^m-point NTT, m rounds of memory data read and write operations are necessary. Using $m = 4$ as an example, Fig. 2 illustrates a GPU-based 16-point NTT implementation. The naive implementation requires four rounds of butterfly operation, which also means four rounds of memory reading and writing, which can be executed by eight threads in parallel in GPU. By merging every two iterations in the naive NTT into a single iteration, the total number of memory reads and writes can be reduced from four rounds to two rounds. With this implementation, the data processed by each thread in each round has increased from two to four. The registers assigned to each thread are sufficient to store these data points, and there is no need for inter-thread synchronization within each round. Even though only half of the original threads are used, the reduction in memory transactions still results in faster NTT processing.

The number of iteration rounds can be condensed even further. However, if the total number of threads is reduced beyond a certain point, the efficiency of NTT processing may decrease because the level of parallelism is also reduced.

Shared-Memory-Level Optimizations: To mitigate memory pressure without further reducing the number of rounds, we expand the NTT implementation into shared memory. Shared memory is an on-chip storage resource that can be shared among threads within the same block, and it offers faster access speeds compared to global memory. We utilize the `__syncthreads()` instruction for thread synchronization within the block, eliminating the need to initiate a new kernel for achieving thread synchronization. Fig. 2 demonstrates the usage of shared memory to decrease global memory access.

When the size of NTT is increased beyond the capacity of shared memory, the entire polynomial cannot be accommodated. For the homomorphic encryption parameter scale (where N is typically between 2^{12} and 2^{17}), N can be decomposed into $N = N_1 \times N_2$, and two kernels are utilized to process NTTs

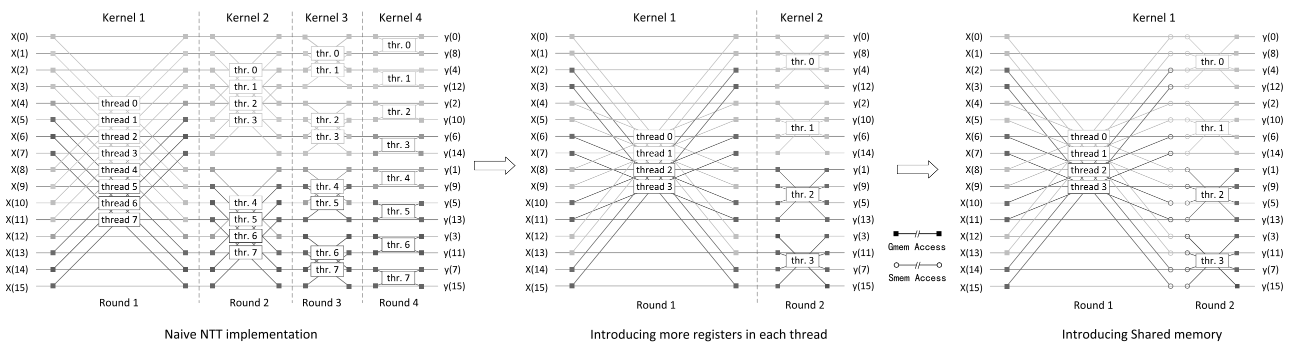


Fig. 2. Reducing GPU kernel count in NTT implementation through the introduction of multiple memory units.

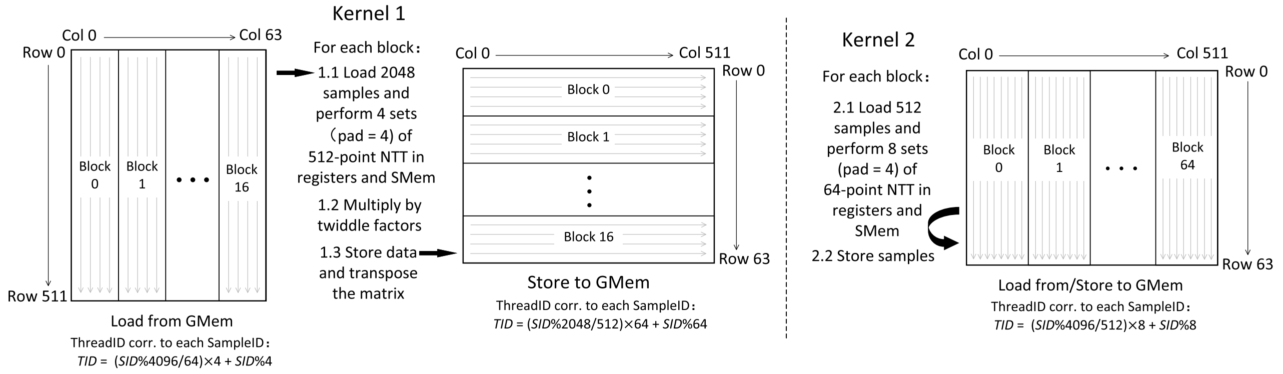

Kernel 1
For each block:
1.1 Load 2048 samples and perform 4 sets (pad = 4) of 512-point NTT in registers and SMem
1.2 Multiply by twiddle factors
1.3 Store data and transpose the matrix
Col 0
Col 63
Row 0
Row 511
Block 0
Block 1
Block 16
Load from GMem
ThreadID corr. to each SampleID:
TID = (SID%4096/64)×4 + SID%4
Col 511
Row 63
Store to GMem
TID = (SID%2048/512)×64 + SID%64
Kernel 2
2.1 Load 512 samples and perform 8 sets (pad = 4) of 64-point NTT in registers and SMem
2.2 Store samples
Block 64
Load from/Store to GMem
TID = (SID%4096/512)×8 + SID%8

Fig. 3. GPU kernels for implementing a 32768-point NTT.

with radix-N_1 and radix-N_2, respectively. This will be further explained in Section V-A2.

Regarding the latest NVIDIA GPUs like the A100 and H100, they support asynchronous data transfers over shared memory, which allows for the overlap of computation and data transfer of multiple data sets [44]. However, to maximize the parallel processing capabilities of GPUs, our approach assigns each block to handle a single data set instead of multiple data sets. Consequently, we do not leverage asynchronous transfer technology in this procedure, ensuring we fully utilize the inherent parallelism of the GPU architecture.

Global-Memory-Level Optimizations: After fully utilizing on-chip resources, the input and output data of the GPU kernels still need to be stored in global memory. Additionally, the twiddle factors are also stored in global memory due to their large size.

To enhance the efficiency of global memory access, we employ coalesced memory access. This approach arranges the data accessed by adjacent threads in neighboring positions, allowing a single memory access to satisfy the data read and write requirements of multiple threads. As a result, the overall number of memory access transactions is reduced.

2) NTT Iterative Scheme and Data Layout: To accommodate the above memory optimizations, we iteratively adopt the 4-step NTT algorithm, dividing the NTT into multiple stages, with smaller NTT sizes. Compared to the high-radix NTT, the inner implementation of the 4-step NTT is simpler, and the access to the twiddle factors is more centralized, making it easier to apply coalesced memory access.

In our implementation, the polynomial length N is initially decomposed into (N_1, N_2), that is, $N = N_1 \times N_2$, where N_1 is close to N_2. Two kernels are employed to execute the 4-step NTT operation: one kernel performs N_1 of N_2-point NTT, while the other kernel performs N_2 of N_1-point NTT.

Fig. 3 illustrates the detailed execution steps of the two kernels using a 32768-point NTT as an example with $N_1 = 512$ and $N_2 = 64$. It depicts the memory access patterns and the execution steps of the kernels. As shown in Fig. 3, each block within Kernel 1 process 4 of N_1-point NTT ($pad = 4$), and each block within Kernel 2 process 8 of N_2-point NTT ($pad = 8$). This ensures that coalesced memory access can always be applied to access adjacent data and provide it to multiple threads.

Within each kernel, each block continues to iteratively use the 4-step NTT algorithm to process the inner NTT. Fig. 4, taking Step 1.1 of Kernel 1 in Fig. 3 as an example, illustrates the specific processing flow of a single block for the 4 sets of 512-point NTT. Specifically, the 512-point NTT is divided into three stages, and each thread processes 8 data points in each stage. Shared memory is utilized for inter-thread data synchronization during this process, and the matrix transpose step is performed simultaneously when writing to shared memory. The processing in Step 2.1 is similar to that in Step 1.1.

3) ModMul Strategy and Negacyclic Convolution: Both Montgomery's ModMul and Shoup's ModMul, implemented in

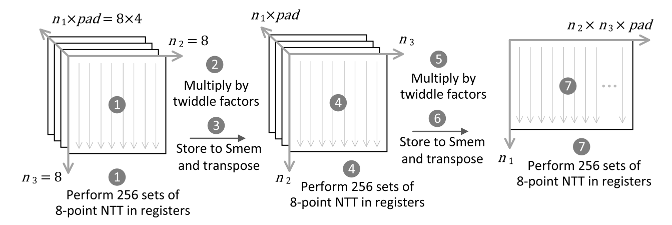

$n_1 \times pad = 8\times4$
$n_2 = 8$
2 Multiply by twiddle factors
3 Store to Smem and transpose
$n_3 = 8$
1
1 Perform 256 sets of 8-point NTT in registers
$n_1 \times pad$
n_3
5 Multiply by twiddle factors
6 Store to Smem and transpose
n_2
4
4 Perform 256 sets of 8-point NTT in registers
$n_2 \times n_3 \times pad$
7 ...
n_1
7 Perform 256 sets of 8-point NTT in registers

Fig. 4. Step 1.1 for each block in NTT kernel 1.

the finite field arithmetic, are applicable to ModMuls within the context of NTT. We evaluated the performance of these two methods in NTT implementation (the results are shown in Section VII-B3), and ultimately, we chose Montgomery's ModMul for its memory-access-friendliness.

Previous works [14], [45] employed a special modulus p_s = 0xFFFFFFFF00000001 to accelerate modular arithmetic by transforming modular reduction into shift and addition/subtraction operations. However, this approach offers limited acceleration benefits and contradicts the negacyclic convolution techniques employed in the NTT algorithm. Consequently, we opted not to utilize this method.

Negacyclic convolution effectively avoids the need for length doubling in NTT operations, resulting in more pronounced improvements. Consequently, we did not employ special moduli but instead chose moduli that satisfy $q_i \equiv 1 \bmod 2N$ to meet the requirements of negacyclic convolution.

In order to improve code reuse and reduce system complexity, this paper applies the negacyclic convolution implementation with pre-processing and post-processing. It is worth noting that the conventional NTT with negacyclic convolution (DWT) requires a pre-processing step during the forward transformation and a post-processing step during the inverse transformation. An alternative approach to implementing the DWT involves combining the pre-processing and post-processing steps with the operations of multiplying twiddle factors. Although this merging technique eliminates the overhead of memory accesses and ModMuls in the pre-processing and post-processing steps, it makes the implementation of the inner layer of DWT more complex and the performance improvement is quite marginal.

B. CRT Optimization With Attentive Handling of Element-Wise Operations

Polynomial arithmetic operations other than NTT, including CRT/ICRT, and operations within the CRT/NTT domain, demonstrate a one-to-one correspondence between inputs and outputs, making them amenable to parallel processing. These operations can be performed by processing the individual terms of polynomials through combinations of underlying mathematical primitives. Based on this, a reasonable GPU thread allocation is necessary to complete a polynomial operation. In order to maximize parallelism, we employ a single thread to process each term in the polynomial.

Within element-wise polynomial operations, CRT and its reconstruction exhibit higher complexity. The choice of algorithms has a considerable impact on implementation performance. Subsequently, we specifically focus on algorithm selection for CRT/ICRT.

1) CRT: In the CRT operation, calculating the remainders of the coefficients is the main computational load. A multi-word integer (MWInt) a is represented by multi-precision representation, i.e., $(a_0, \ldots, a_{m-1})$, such that $a = \sum_{j=0}^{m-1} a_j \beta^j$. The moduli are single-word integers. There are two ways to implement CRT transform, using different methods for the multi-word integer modulo single-word integer (MSMod) operation, referred to as CRT-MOD and CRT-MUL, respectively.

CRT-MOD: The fist method follows the basic long division method, which calculates each term iteratively using the formula $h_j = (h_{j-1} \cdot \beta + a_{m-1-j}) \bmod q_i$ for $j \in 1 \le j \le m-1$, with the h_0 being initialized as $h_0 = a_{m-1}$, and h_{m-1} is the final remainder obtained from the operation. However, the native modulo operation (%) on GPUs exhibits poor efficiency. To overcome this limitation, we use the Barrett reduction method [41] to speed up the modulo operations.

CRT-MUL: The other approach to computing MSMod is based on the idea of precomputation. The word size β and the modulus q_i are independent of the input, thus we can precompute $\beta^j \bmod q_i$ for all j as sequence $\mathbf{b}$. In this way, an MSMod is transformed into the inner product of two sequences: $a \bmod q_i = \sum_{j=0}^{l-1} a_k \cdot \beta^j \bmod q_i = \sum_{j=0}^{l-1} a_j \cdot (\beta^j \bmod q_i) \bmod q_i = \mathbf{a} \cdot \mathbf{b} \bmod q_i$. In our implementation, the inner product is performed by a batch of SWMuls and a summation of products. The optimization of MWAcc is still adopted here and Barrett reduction is used for the modulo operation of the summation.

2) ICRT: Compared with CRT, CRT reconstruction (ICRT) is more complicated. Two implementations of ICRT are developed, adopting the classic CRT algorithm and Garner's algorithm [46] respectively. To apply the optimized operations in underlying arithmetic, we decouple the main computational loads of the two algorithms.

ICRT-C: The classic ICRT runs basically as (1) in Section II-C1 indicates. The main computational loads involve obtaining the accumulated sum with MWMAC and taking its modulus with respect to Q using MWMod. The MWMAC is analyzed in Section IV-C, while the MWMod in ICRT can be achieved by an MWCom and followed by an MWSub if needed. To reduce the computational overhead at runtime, we precompute $Q, q_i, ratio_i, Q/q_i$ and $(Q/q_i)^{-1} \bmod q_i$ for the classic ICRT algorithm and store them in the constant memory of GPU.

ICRT-G: Another CRT reconstruction approach, adopting Garner's algorithm [46], is shown in Algorithm 2. We precompute $k_i = \prod_{j=0}^{i-1} q_j$ and $c_i = \prod_{j=0}^{i-1} (q_j^{-1} \bmod q_i) \bmod q_i$ for $i \in \{1 \le i < r\}$ and store them in the constant memory. ICRT-G has different computational loads from ICRT-C. Although it also includes the MWMAC operation in Line 7, which involves multiplying k_i by u and accumulating the result onto a, the average word number of k_i is obviously shorter than that of Q/q_i in ICRT-C method. Besides, there are MSMod operations in Line 5, for which we employ the method used in CRT-MUL implementation.

Algorithm 2: ICRT Using Garner's Algorithm.

```
Require: $r$ remainders representing integer $a$ ($a < p$) in
  CRT domain: $\{a_0, a_1, \ldots, a_{r-1}\}$
Ensure: the integer $a$
1: $a = a_0$
2: $u = (a - a_1) \cdot c_1 \bmod q_1$
3: $a = a + u \cdot k_1$
4: for $i = 2$ to $r - 1$ do
5:     $u = \texttt{MSMod}(a, q_i)$            ▷ $u = a \bmod q_i$
6:     $u = (a_i - u) \cdot c_i \bmod q_i$
7:     $\texttt{MWMAC}(k_i, u, a)$              ▷ $a += u \times k_i$
```

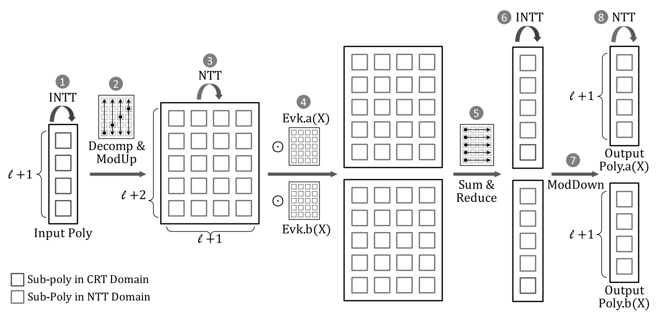


Fig. 5. The key-switching procedure.

VI. Homomorphic Operations

Taking the CKKS scheme as a typical research case, we implement homomorphic operations within the CKKS scheme based on our polynomial arithmetic implementation. To enhance the performance of our implementation, we employ device-level acceleration techniques. This involved parallelizing and fusing multiple kernels, as well as employing pre-transmission and pre-allocation techniques to reduce memory transmission and allocation overhead. In this section, we elaborate on the kernel enhancement and memory management techniques employed in homomorphic operations. Furthermore, we demonstrate the implementation of a representative homomorphic operation, key-switching, and highlight the application of two kinds of optimization techniques within it.

A. Kernel Enhancement

The implementation of the homomorphic operations typically involves multiple GPU kernels. To enhance their computational efficiency, we employ two pivotal techniques: kernel consolidation and kernel fusion.

First, in optimizing homomorphic operations on multiple polynomials, we identify substantial potential for parallel execution. While a typical approach executes each loop iteration as a separate CUDA kernel, this doesn't fully utilize available parallelism. One potential solution uses multiple CUDA streams to achieve dynamic load balancing with minimal code changes, but the scheduling overhead and synchronization dependencies led us to explore alternatives Instead, we propose Kernel Consolidation by unifying iterative processes into a singular, expansive computational kernel. This is achieved by meticulously redesigning the control flow, aligning data structures and parallel execution paths to optimize memory access and resource utilization. A unified kernel reduces the overhead of frequent launches and provides opportunities for more granular optimizations, such as improved memory access patterns and shared resource usage. Our implementation strategically reorganizes computation, leveraging thread and block hierarchies to maximize throughput and scalability on modern GPU architectures.

Additionally, continuous operations on the same polynomial are required in homomorphic operations and are implemented as separate kernels at the polynomial arithmetic level. To streamline this process, when there is no need for inter-thread data synchronization, we employ kernel fusion to combine multiple individual kernels into a larger kernel and utilize registers to store intermediate data. In GPU-based parallel design work, kernel fusion generally refers to merging various computational tasks or types of compute operations within a single kernel [47], [48]. In homomorphic encryption research, it often denotes the fusion of consecutive kernels [16], [19]. We apply this technique to our polynomial arithmetic kernels to reduce the number of kernels, thereby lowering kernel launch overhead and global memory access, resulting in enhanced computational efficiency.

B. Memory Management

Homomorphic operations often require handling large-scale data. To minimize waiting times for GPU kernels and ensure seamless execution streams, we implement efficient memory management.

First, it is essential to perform precomputation and pre-transmission. During the initialization phase, we proactively transfer precomputed data used in homomorphic operations to the GPU memory. This data includes a series of keys, moduli and their variations, and twiddle factors of the NTT/INTT operation. This preemptive step reduces data transfer latency and minimizes bandwidth consumption.

Moreover, within homomorphic operations, data may undergo multiple copying and expansion processes. To streamline this, we introduce a memory pool for advanced memory allocation. During the computation phase, the program directly accesses appropriately sized memory from the memory pool, significantly reducing memory allocation latency.

C. Key-Switching

Serving as the main workload in `HROTATE` and `HMULT`, the key-switching operation has a significant impact on the performance of most homomorphic applications. Our key-switching implementation utilizes the algorithm proposed in the previous work [31] and employs a commonly used `dnum` setting with $\texttt{dnum} = L + 1$. The workflow of key-switching is illustrated in Fig. 5. In this process, we employ kernel enhancement and memory management techniques to enhance computational efficiency.

Regarding kernel consolidation, for both the second and third steps, we integrate the processing of each RNS-decomposition component into broader kernels. In steps six through eight, we

TABLE II
EXPERIMENTAL PLATFORM CONFIGURATION

GPU	A100-PCIE-80G @1.41Ghz	Tesla P100-PCIE-16G @1.19Ghz
CPU	Hygon C86 7265 @2.18GHz	Intel Core i9-10940X @3.30GHz
OS	AliOS 7.2	Ubuntu 20.04
CUDA Toolkit	CUDA 11.3	

continue utilizing a single kernel at each step to process both the upper and lower sections. This approach enables more efficient GPU utilization by fully expanding computations. Concerning kernel fusion, for the intermediate fourth and fifth steps, which are not involved in NTT computations and are primarily memory-bound operations, we merge them into a single kernel. This fusion reduces redundant memory accesses and minimizes the overhead associated with writing intermediate results to global memory and subsequently reading them again.

In terms of memory management, in the 4th step of key-switching, high-dimensional evaluation keys are introduced as input. We have pre-transferred this data to GPU memory during the initialization phase. Additionally, the data generated in the 2nd and 4th steps exceeds the input data size by several times. It is essential to minimize noticeable time overhead in memory allocation. Thus, we allocate the corresponding memory in advance within the memory pool, and during the computation phase, we simply retrieve the memory pointers from the memory pool.

VII. EVALUATION AND DISCUSSION

In this section, we evaluate the effectiveness of different underlying arithmetic and multi-level optimization methods and compare them with previous works. To demonstrate in detail how we achieve performance advantages, we evaluate the performance of GIF-FHE at multiple implementation layers, including polynomial arithmetic, homomorphic operations, and homomorphic applications.

A. Experimental Setup

The experiments are conducted on two servers, one equipped with NVIDIA A100 GPU and the other equipped with NVIDIA Tesla P100 GPU, respectively. The A100 GPU represents GPUs that strike a balance between floating-point and integer computation capabilities, while the P100 GPU stands out for GPUs that allocate more resources to floating-point computation. Although the latest H100 GPU has floating-point computation capabilities that are twice as powerful as its integer computation capabilities, it is currently challenging to obtain a server equipped with the H100 GPU. Therefore, we utilized the P100 GPU as a representative of this type of GPU. The specific hardware and software configurations in the experiments are detailed in Table II.

In this section, we express the polynomial parameters in a simplified form, denoted as ($\log N$, $\log Q$, r), e.g., (15, 881, 16).

B. Evaluation of Polynomial Arithmetic

GIF-FHE offers several optimization algorithms and three underlying arithmetics for polynomial arithmetic. Experimental assessments are conducted on the performance of CRT/ICRT and NTT/INTT implementations using different algorithms and underlying arithmetics.

TABLE III
PERFORMANCE (μs) OF NTT, INTT, AND ELEMENT-WISE MULTIPLICATION (EWM) WITH PARAMETERS OF $(15, 881, 18)$ IN OUR WORK COMPARED TO CPU IMPLEMENTATIONS AND SPEEDUP S1 (GIF-FHE ON P100 VERSUS NATIVE C++) AND S2 (GIF-FHE ON A100 VERSUS NATIVE C++)

Op.	HEXL [50] Native C++ (i9-10940X)	HEXL [50] AVX512-DQ (i9-10940X)	GIF-FHE FP64 (P100)	GIF-FHE INT64 (A100)	S1	S2
NTT	6678	2358	95.4	38.6	70×	173×
INTT	6734	2520	96.2	39.4	70×	171×
EWM	818	463	29.2	10.6	28×	77×

TABLE IV
PERFORMANCE (μs) OF CRT IMPLEMENTATIONS AND SPEEDUP S (GIF-FHE VERSUS HE-BOOSTER)

Und. Arith.	Param.	Op.	HE-Booster (3070)	GIF-FHE (A100)	GIF-FHE (P100)	S
INT32	14,448,16	CRT	40.0	14.2	24.7	2.8×
		ICRT	78.0	27.3	72.1	2.9×
	15,896,32	CRT	219.8	36.9	145.2	6.0×
		ICRT	431.8	93.5	443.3	4.6×
INT64	14,448,16	CRT	21.0	13.6	26.3	1.5×
		ICRT	69.5	20.3	47.6	3.4×
	15,896,32	CRT	126.4	32.7	137.3	3.9×
		ICRT	337.8	54.1	259.7	6.2×

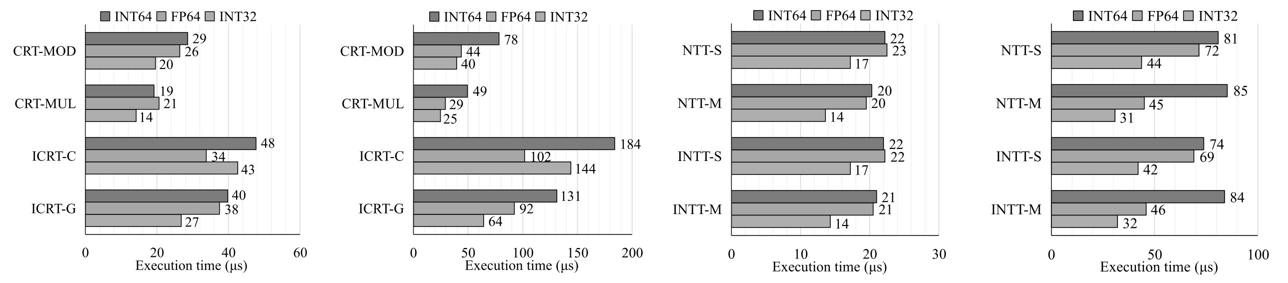
TABLE V
PERFORMANCE (μs) OF NTT IMPLEMENTATIONS WITH INT32 ARITHMETIC ON P100 GPU AND SPEEDUP S1 (GIF-FHE VERSUS CUHE), S2 (GIF-FHE VERSUS [51]) AND S3 (GIF-FHE VERSUS K3)

Param.	Op.	cuHE [51]	[52]	K3*	GIF-FHE	S1	S2	S3
13,372,13	NTT	48.5	40.0	19.1	16.2	3.0×	2.5×	1.2×
	INTT	53.4	48.0	20.9	17.7	3.0×	2.7×	1.2×
14,744,25	NTT	160.9	87.0	52.3	43.2	3.7×	2.0×	1.2×
	INTT	183.1	93.0	55.8	43.9	4.2×	2.1×	1.3×

*: K3 refers to the NTT/INTT implementation that uses three kernels instead of two, in comparison with GIF-FHE.

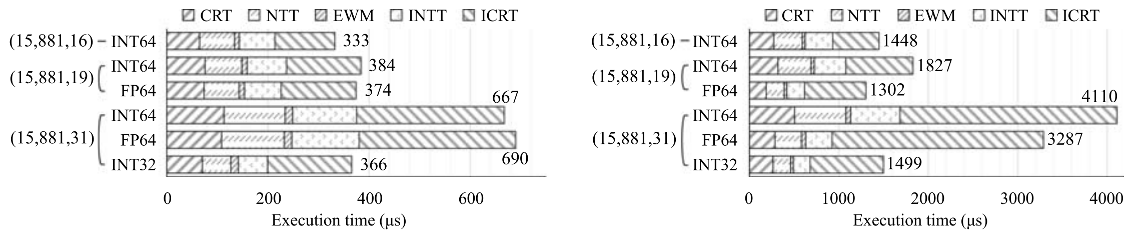
1) Parameter Selection: Figs. 6 and 7 show the impact of different algorithms and underlying arithmetic on polynomial arithmetic performance, with results for $\log N = 14$ and $\log N = 15$, respectively. Variations in polynomial degrees have minimal effect on the conclusions. In the experiments shown in Fig. 6, we use polynomial parameters of $(14, 438, 16)$ with modulus sizes less than 29 bits to ensure compatibility with all three arithmetic methods. In Fig. 7, for the comparison of the three underlying arithmetic, we employ a wider range of modulus sizes to fully demonstrate their differences. Meanwhile, the experiments presented in Tables III, IV, and V utilize parameter settings from the target articles to ensure a fair comparison with their implementations.

2) CRT/ICRT Evaluation: We compare two CRT implementations, employing algorithms CRT-MOD and CRT-MUL, respectively. Fig. 6(a) and (b) show the results of CRT/ICRT implementations. For the CRT algorithm, CRT-MUL performs better than CRT-MOD in all implementations with different arithmetic. Both of them transform the MSMod operation to single-word multiplications by Barrett reduction or precomputation. Although the precomputed table takes up a certain amount of constant memory, it does reduce the number of multiplications and gains obvious benefits.



(a) CRT/ICRT Impl. on A100 (b) CRT/ICRT Impl. on P100 (c) NTT/INTT Impl. on A100 (d) NTT/INTT Impl. on P100

Fig. 6. Execution time of CRT and NTT implementations with parameters of $(14, 438, 16)$ on A100 and P100.



(a) Impl. on A100 (b) Impl. on P100

Fig. 7. Execution time of a polynomial multiplication on A100 and P100.

ICRT is more time-consuming than CRT. Two implementations of ICRT are evaluated, using ICRT-C and ICRT-G, respectively, Garner's algorithm improves the performance of ICRT substantially in most implementations with different underlying arithmetic. On the whole, the average size of MWMul multipliers in Garner's algorithm is smaller than that in classic ICRT. Although several MSMods are required, the overall cost of ICRT-G is still less than that of ICRT-C after the optimizations of main computing loads.

It can be noticed that the ICRT-C algorithm demonstrates superior performance when employing FP64 arithmetic compared to the other two arithmetic options, primarily due to the efficient implementation of MWMAC in FP64 arithmetic. Within ICRT-C, the MWMAC operation occupies a substantial proportion of time, and the FP64 arithmetic's MWMAC implementation minimizes the need for carry operations, resulting in prominent instruction savings. In cases where the moduli are small, the implementation of the ICRT-C algorithm using FP64 arithmetic exhibits superior performance compared to the ICRT-G algorithm on the A100 platform. However, as the modulus increases, the ICRT-C algorithm experiences an increase in computational complexity and memory access, whereas the ICRT-G algorithm demonstrates a negligible increase in execution time. When the moduli approach 49 bits, the implementation utilizing Garner's algorithm remains obviously faster than the conventional ICRT implementation.

Additionally, in the CRT-MUL method, the implementation utilizing FP64 arithmetic exhibits slower performance compared to the implementation employing INT64 arithmetic on the A100 platform. This discrepancy arises from the fact that the `mad` instruction can be utilized in integer arithmetic, whereas in FP64 arithmetic, multiplication and addition must be separate instructions.

3) NTT/INTT Evaluation: For NTT/INTT, we compare two setups that exploit Shoup's ModMul (NTT/INTT-S) and Montgomery's ModMul (NTT/INTT-M), respectively. They both adopt negacyclic convolution with pre-processing and post-processing.

The result on A100 shows that Montgomery's ModMul is 5%-18% faster than Shoup's ModMul in implementations with different arithmetics. This indicates that Montgomery's ModMul with fewer memory accesses is more suitable for NTT/INTT than Shoup's ModMul. As to the implementations on P100, Montgomery's ModMul in INT64 arithmetic takes more time compared to Shoup's ModMul. However, in FP64 arithmetic and INT32 arithmetic, the advantage of Montgomery's ModMul can reach up to 30% or even higher. This demonstrates that Montgomery's ModMul excels more prominently in scenarios with high computational throughput in hardware.

4) Analysis of Different Arithmetics: To further illustrate the effectiveness of different underlying arithmetics, we evaluate the polynomial multiplication performed starting from the polynomial domain with parameters $(\log N = 15, \log Q = 881)$. The operation encompasses CRT, NTT, element-wise multiplication (EWM), INTT, and ICRT operations of two polynomials. We utilize the optimization algorithms that perform better in previous tests, namely CRT-MUL, ICRT-G, NTT-M, and INTT-M. The outcomes of this assessment are presented in Fig. 7.

In this experimental setup, we select three values for r (16/19/31) to emphasize the distinctive features of various underlying arithmetic. When $r = 16$, the moduli are set to $54 \sim 56$ bits, resulting in a lower supported multiplication depth but higher computational precision. When $r = 31$, the moduli are set to $27 \sim 29$ bits, leading to a higher supported multiplication depth but lower computational precision. In our implementation, INT64 arithmetic supports all the three choices of moduli, FP64 arithmetic supports $r \geq 19$, and INT32 arithmetic supports $r \geq 31$.

In the experiments conducted on the P100 GPU, it is observed that FP64 arithmetic achieves substantially faster speeds compared to INT64 arithmetic with the same parameters. For the polynomial multiplication, FP64 arithmetic demonstrates a performance improvement ranging from 25% to 40%. Specifically, for the NTT/INTT operation, FP64 arithmetic achieves an improvement range of 80% to 90%. Under the parameter setting of $r = 31$, INT32 arithmetic exhibits faster speeds. The NTT/INTT performance of INT32 arithmetic is found to be $2.93/2.96\times$ faster than INT64 arithmetic. Additionally, for the polynomial multiplication, the INT32 implementation is $2.74\times$ faster than the INT64 implementation.

On the A100 GPU, INT64 arithmetic and FP64 arithmetic exhibit similar performance. However, when the parameter r is set to 31, INT32 arithmetic still maintains a performance advantage. Specifically, the NTT/INTT implementation using INT32 arithmetic exhibits a speed improvement of $2.14/2.17\times$ compared to the INT64 implementation. Additionally, there is an 82% performance improvement observed for the polynomial multiplication operation.

To summarize, there are several key observations. First, the performance advantage of FP64 arithmetic is evident on GPUs, such as the P100, that prioritize floating-point arithmetic, and we believe it also holds for H100. Second, INT32 arithmetic demonstrates considerable advantages over INT64 arithmetic on different GPUs, as it requires fewer instructions for computations. On the other hand, while INT64 arithmetic may not stand out in terms of performance, it offers support for the widest range of moduli. These findings highlight the trade-offs and considerations when selecting underlying arithmetic for specific parameters and GPU architecture.

5) Comparison With CPU Implementations: In Table III, we compare our GPU-based implementations with the state-of-the-art CPU implementations. Compared to the native C++ kernel in the HEXL library [49], our INT64 implementation on A100 delivers a performance boost of over $20\times$ when processing the NTT/INTT operation of a single polynomial.

Thanks to the GPU's massively parallel architecture and pipeline mechanism, multiple polynomials can be processed in parallel. For the commonly used parameters, (15,881), our INT64 implementation on the A100 GPU requires only 38.6/39.4 μs to calculate 18 sets of polynomial components. This is $173/171\times$ faster than the native C++ implementation. Compared to optimized implementations using AXV512-DQ instructions, our implementation still achieves a performance improvement of $61/64\times$.

More advanced CPUs support AVX512-IFMA implementation, which is twice as fast as AVX512-DQ implementation [13], but even so, our A100-based implementation still has an obvious performance advantage.

6) Comparison With GPU Implementations: In order to evaluate the performance of our CRT and NTT implementations, we conducted a comparison with state-of-the-art GPU implementations. Specifically, we compared the performance of our CRT/ICRT implementation with that of HE-Booster [18], as presented in Table IV. Our implementation, utilizing INT32 arithmetic and INT64 arithmetic on the A100, demonstrates performance improvements of $2.8 \sim 6.0\times$ and $1.5 \sim 6.2\times$, respectively. It's worth noting that the A100 GPU we used and the 3070 GPU employed by HE-Booster both belong to the Ampere architecture. While the 3070 has a higher frequency and the A100 has more cores, their computational capabilities are very close (20.31 and 19.49 TFLOPS). Since HE-Booster also incorporates Barrett reduction and the Garner's algorithm, we infer that the performance improvements primarily result from our meticulous optimization of the underlying arithmetic and the efficient utilization of precomputations.

Regarding the NTT, we conducted comparisons between our implementation and cuHE [14], [50] as well as the work by Al Badawi et al. [51], as presented in Table V. The experiments are performed on an NVIDIA P100 GPU, maintaining consistency with the hardware used in the previous work [51]. cuHE stands as an open-source homomorphic encryption library for GPUs and, similar to our approach, employs the 4-step NTT technique. Compared to cuHE, the NTT method in GIF-FHE enhances performance by $3.0 \sim 4.2\times$ on the same GPU (P100) through efficient modulus operations, the introduction of negacyclic convolution, and reducing the number of kernels from 3 to 2. In Table V, we also compare an implementation, referred to as K3, which utilizes the same underlying arithmetic but implements a more granular NTT decomposition using 3 kernels. As the number of kernels decreases, global memory access is also reduced. This optimization leads to a performance improvement of the NTT in GIF-FHE by a factor of $1.2 \sim 1.3\times$. On the other hand, another work [51] adopts a variant of NTT called Discrete Galois Transform (DGT). Our implementation demonstrates a performance advantage of $2.0 \sim 2.7\times$ compared to their approach.

Additionally, HE-Booster [18] discloses the performance of individual NTT operations. Compared to their NTT performance, our NTT exhibits lower latency. For instance, with the parameter $\log N = 15$, our NTT/INTT implementation utilizing INT64 arithmetic achieves a latency of 15.4/16.3 μs, surpassing their performance ($19 \sim 20\mu s$). Compared to previous works, the performance advantage of our NTT implementation stems from the efficient utilization of multi-level GPU memory and the meticulous implementation of finite field arithmetic operations.

We also compare our method with TensorFHE [52], a solution based on tensor cores that primarily optimizes throughput. Although the primary focus of this paper is on reducing the latency of homomorphic operations on single ciphertexts, we conduct throughput experiments with parameters (12, 108, 4), (13, 217, 8), and (14, 437, 16), using batch sizes of 128, 64, and 32, respectively. The achieved NTT throughput is 8.13, 3.80, and 1.85 million operations per second, which is 8.9/8.4/8.8$\times$ higher than the NTT throughput in TensorFHE. This significant performance improvement is primarily due to our single NTT/INTT operations requiring only two kernels, whereas TensorFHE involves five types of kernels from stage 1 to stage 5 due to bit decomposition and merging. GIF-FHE significantly reduces off-chip memory read and write operations by fully utilizing on-chip memory and reducing the number of kernels, thereby greatly enhancing the performance of NTT/INTT.

TABLE VI
PERFORMANCE COMPARISON OF KEY-SWITCHING IMPLEMENTATIONS ON A100

	Impl.	Execution time (ms)	Speedup vs [54]		Impl.	Execution time (ms)	Speedup vs [54]		Impl.	Execution time (ms)	Speedup vs [54]
SET-A1 (13,218,8)	100x [54]	0.28		SET-A2 (13,218,5)	100x [54]	0.21		SET-A3 (13,218,4)	100x [54]	0.19	
	INT64	0.10	2.80×		INT64	0.08	2.63×		INT64	0.07	2.71×
	FP64	0.10	2.80×		FP64	0.08	2.63×				
	INT32	0.07	**3.50×**								
SET-B1 (14,438,16)	100x [54]	0.62		SET-B2 (14,438,9)	100x [54]	0.34		SET-B3 (14,438,8)	100x [54]	0.31	
	INT64	0.30	2.07×		INT64	0.14	2.43×		INT64	0.13	2.38×
	FP64	0.29	2.14×		FP64	0.14	2.43×				
	INT32	0.26	2.38×								
SET-C1 (15,881,31)	100x [54]	2.45		SET-C2 (15,881,18)	100x [54]	1.03		SET-C3 (15,881,16)	100x [54]	0.86	
	INT64	1.75	1.40×		INT64	0.66	1.56×		INT64	0.54	1.59×
	FP64	1.64	1.49×		FP64	0.60	1.72×				
	INT32	0.54	1.59×								

TABLE VII
PERFORMANCE COMPARISON OF KEY-SWITCHING IMPLEMENTATIONS ON P100

	Impl.	Execution time (ms)	Speedup vs [54]		Impl.	Execution time (ms)	Speedup vs [54]		Impl.	Execution time (ms)	Speedup vs [54]
SET-A1 (13,218,8)	100x [54]	0.59		SET-A2 (13,218,5)	100x [54]	0.44		SET-A3 (13,218,4)	100x [54]	0.39	
	INT64	0.52	1.13×		INT64	0.40	1.12×		INT64	0.38	1.03×
	FP64	0.19	3.11×		FP64	0.14	3.14×				
	INT32	0.19	3.11×								
SET-B1 (14,438,16)	100x [54]	2.58		SET-B2 (14,438,9)	100x [54]	1.10		SET-B3 (14,438,8)	100x [54]	1.01	
	INT64	1.94	1.33×		INT64	0.94	1.17×		INT64	0.89	1.13×
	FP64	0.79	3.27×		FP64	0.33	**3.33×**				
	INT32	0.79	3.27×								
SET-C1 (15,881,31)	100x [54]	12.91		SET-C2 (15,881,18)	100x [54]	4.52		SET-C3 (15,881,16)	100x [54]	3.91	
	INT64	11.04	1.17×		INT64	4.02	1.12×		INT64	3.30	1.18×
	FP64	4.22	3.06×		FP64	1.60	2.83×				
	INT32	4.29	3.01×								

C. *Evaluation of Homomorphic Operations*

Building upon the groundwork of underlying arithmetic and polynomial arithmetic, we further implement homomorphic operations to conduct a more comprehensive assessment of performance. For the convenience of application usage, we uniformly use the INT64 type for data storage at the homomorphic operation layer, converting it to the required data type during underlying arithmetic computations.

We initially compare our implementation with the fastest existing CUDA core-based implementation [16] (we denote it as the 100x implementation). The 100x implementation in our experiments utilized the open-source code from this work [53]. Due to the incompleteness of the open-source code implementation, our comparison focused on the key-switching operation. For parameter selection, we use three sets of N values ranging from 2^{13} to 2^{15} to cover diverse scenarios, each paired with a corresponding Q. To highlight characteristics of different underlying arithmetics (INT64, FP64, and INT32), we assess performance across various implementations for each N, Q combination. For instance, with $(\log N = 15, \log Q = 881)$, we set $r = 31/18/16$. Under the configuration where $\mathtt{dnum} = l + 1$, the supported levels of homomorphic multiplication (l) are $29/16/14$.

The results are presented in Tables VI and VII. For the 100x implementation, the *GRID_NUM* for the NTT kernels is set to 2048, the *FIRST_STAGE_RADIX_NUM* is set to 256 for $\log N$ values of 14 and 15, and set to 128 when $\log N$ is 13. On the A100 platform, by combining polynomial arithmetic optimization with further enhancements in kernel compactness, the performance of the INT64 implementation achieves up to a $3.11\times$ improvement compared to the 100x. Kernel consolidation more effectively leverages increased parallelism with smaller parameters, resulting in relatively higher speedup ratios for small parameters. On the A100, the performance of the FP64 implementation is similar to that of the INT64 implementation. However, when the modulus is small, employing INT32 arithmetic can enhance the performance of the key-switching implementation to approximately $3.5\times$ that of the 100x implementation.

On the P100 platform, the floating-point unit-based implementation demonstrates significant performance advantages. Under the set-A2/B2/C2 configurations, the FP64 version exhibits performance improvements ranging from $2.51\times$ to $2.86\times$ compared to the INT64 version and from $2.83\times$ to $3.33\times$ compared to the 100x implementation. Due to the higher computational complexity of key-switching compared to individual polynomial multiplications, the advantages of FP64 arithmetic are more pronounced in the key-switching experiments. Due to the relatively weak computational capabilities for integer numbers, the INT64 implementation using NTT-M shows a modest improvement compared to the 100x implementation.

Additionally, we further implement `HMULT` and `HROTATE` in the CKKS scheme, and conduct performance comparisons with the SEAL library and the state-of-the-art GPU implementation, HE-Booster [18]. We use the same parameters as HE-Booster for consistency. The results are illustrated in Table VIII. Our

TABLE VIII
PERFORMANCE (MS) OF HMULT AND HROTATE IMPLEMENTATIONS AND GIF-FHE'S SPEEDUP OVER OTHER IMPLEMENTATIONS

Impl.	Device	TFLOPS*	HMULT (13,218,4)	HMULT (14,448,8)	HMULT (15,881,16)	HROTATE (13,218,4)	HROTATE (14,448,8)	HROTATE (15,881,16)
SEAL [10]	i9-10940X	/	3.22	24.18	184.42	2.72	22.54	172.53
HE-Booster [18]	RTX3070 GPU	20.31	0.21	0.58	2.22	0.18	0.47	2.04
GIF-FHE (FP64)†	P100 GPU	9.53	0.18	0.39	1.72	0.22	0.48	2.69
Baseline (INT64)	A100 GPU	19.49	0.51	1.92	6.32	0.53	2.03	6.93
MemMgmt (INT64)			0.15	0.31	1.11	0.14	0.31	1.09
GIF-FHE (INT64)			0.09	0.16	0.58	0.08	0.15	0.57
Speedup vs [10]			36.2×	151.1×	318.5×	33.2×	155.5×	304.8×
Speedup vs [18]			2.36×	3.63×	3.83×	2.20×	3.24×	3.60×

*: The TFLOPS (Trillion Floating-point Operations Per Second) listed here represents the nominal single-precision capability. Note that the real-time TFLOPS performance of a GPU fluctuates within acceptable limits as the voltage and temperature change.
†: The parameters used by GIF-FHE (FP64) are (13, 218, 5), (14, 448, 9), (15, 881, 18).

TABLE IX
PERFORMANCE OF HE-CNN INFERENCE ON MNIST

Impl.	Platform	HOP	KS	λ	N	Q	Lat. (s)	Scheme
CryptoNets [55]	Intel Xeon E5-1620L	215K	945	-	-	-	205	BFV
AHEC [56]	Xeon Platinum 8180 with 112 CPUs	215K	945	128	13	-	29.17	CKKS
LoLa [57]	Azure B8ms VM with 8 vCPUs	798	227	128	14	440	2.2	BFV
FxHENN [58]	ALINX ACU15EG	826	280	128	13	210	0.19	CKKS
A*FV [59]	$3 * $P100 and $1 * $V100GPUs	47K	0	82	13	330	5.2	BFV
HE-Booster [18]	NVIDIA 3070 GPU	215K	945	128	13	210	4.2	BGV
GIF-FHE (INT32)	NVIDIA A100 GPU	826	280	128	13	210	**0.044**	CKKS

implementation employs the same algorithm and dnum setting as the SEAL library, achieving a performance that is $318.5\times$ and $304.8\times$ higher than SEAL for HMULT and HROTATE when configured with parameters $(\log N = 15, \log Q = 881, r = 16)$. Due to optimizations in our underlying arithmetic and polynomial operations, our implementation achieves a performance $3.83/3.60\times$ higher than HE-Booster under the parameters $(15, 881, 16)$. Leveraging the optimizations in FP64 arithmetic, our implementation on the relatively less powerful P100 GPU can achieve performance levels close to that of the HE-Booster implementation.

We also evaluate the effects of kernel enhancement and memory management strategies in Table VIII. This evaluation uses INT64 arithmetic and the A100 GPU. The implementation using pre-transfer and memory pooling (MemMgmt) shows a $3.5 \sim 7.0\times$ performance improvement compared to the implementation using only pre-transfer (Baseline), mainly due to reduced overhead in memory allocation and deallocation. Further enhancements with kernel consolidation and kernel fusion (GIF-FHE) provide an additional $1.6 \sim 2.1\times$ performance boost, because of increased parallelism and reduced unnecessary memory accesses.

D. Evaluation With HE Applications

Homomorphic encryption-enabled convolutional neural network (HE-CNN) inference on encrypted data stands out as a typical application of homomorphic encryption. In Table IX, we present representative CPU implementations, state-of-the-art GPU implementations, and FPGA implementations for HE-CNN inference on the single image from MNIST [59] dataset. Lola [56] is a representative solution for implementing HE-CNN inference on CPUs. It reduces the number of homomorphic operations ("HOP") to 798 and key-switching operations ("KS") to 227, resulting in a reduced inference time of 2.2 seconds per image. FxHENN [57], which employs a similar number of operations as Lola, utilizes the CKKS scheme and achieves a further reduction in inference time to 0.19 seconds per image on the ALINX FPGA platform. Notably, existing GPU implementations of the HE-CNN inference solution incur a higher count of HOP and KS. As of now, HE-booster [18] achieves the fastest GPU-based HE-CNN inference for individual images from the MNIST dataset, with a processing time of 4.2 seconds.

We utilize the same HE-CNN model as FxHENN, comprising five layers, including convolutional layers, activation layers, densely connected layers, and activation layers. Built upon the homomorphic operations of GIF-FHE, we reduce the inference latency on a single MNIST image to 0.044 seconds, which is $49.8\times$ faster than Lola. Compared to the solution provided by HE-Booster, GIF-FHE has advantages in both homomorphic operation performance and the HOP and KS required, achieving $95.5\times$ the performance. Compared to the fastest FPGA-based solution, FxHENN, GIF-FHE achieves $4.3\times$ the performance for a single image inference due to our faster homomorphic operations.

VIII. CONCLUSION

In this contribution, we leverage multiple computing resources of GPUs to accelerate the expensive FHE operations. A full set of low-level and middle-level FHE primitives is implemented based on two arithmetic units with three types of data precision. A case study with CKKS as an example demonstrates that all three series of our implementations surpass the state-of-the-art GPU-based implementation, with performance gains of up to $3.8\times$. Moreover, our solution outperforms the mainstream CPU library, SEAL, by more than $300\times$. The detailed evaluation and comparison of this paper would offer a vital reference for the follow-up work to choose an appropriate road-map in GPU-based FHE implementations. Our future work will further improve the fundamental primitives, extend our work to other FHE schemes (e.g., TFHE [6]) and apply the GPU-based FHE schemes to more real-world workloads.

VIII. DISCLOSURE

A preliminary version of this paper appeared under the title Towards Faster Fully Homomorphic Encryption Implementation with Integer and Floating-point Computing

Power of GPUs, in Proc. 2023 IEEE 37th International Parallel and Distributed Processing Symposium, St Petersburg, Florida, May 15-19, 2023 [60].

References

[1] R. L. Rivest et al., "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 169–178.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, 2014.

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, 2012, Art. no. 144.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, Springer, 2017, pp. 409–437.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[7] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A graph compiler for deep learning on homomorphically encrypted data," in *Proc. 16th ACM Int. Conf. Comput. Front.*, 2019, pp. 3–13.

[8] G. Onoufriou, M. Hanheide, and G. Leontidis, "EDLaaS: Fully homomorphic encryption over neural network graphs," 2021, *arXiv:2110.13638*.

[9] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A high-throughput framework for neural network inference on encrypted data," in *Proc. 7th ACM Workshop Encrypted Comput. Appl. Homomorphic Cryptography*, 2019, pp. 45–56.

[10] Microsoft, "Microsoft SEAL (release 4.0)," 2022. Accessed: Aug. 2024. [Online]. Available: https://github.com/Microsoft/SEAL

[11] HElib, 2022. Accessed: Aug. 2024. [Online]. Available: https://github.com/homenc/HElib

[12] V. Sidorov, E. Y. F. Wei, and W. K. Ng, "Comprehensive performance analysis of homomorphic cryptosystems for practical data processing," 2022, *arXiv:2202.02960*.

[13] F. Boemer, S. Kim, G. Seifu, F. D. de Souza, and V. Gopal, "Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52," 2021, *arXiv:2103.16400*.

[14] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Proc. Int. Conf. Cryptogr. Inf. Secur. Balkans*, Springer, 2015, pp. 169–186.

[15] W. Jung et al., "Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *IEEE Access*, vol. 9, pp. 98772–98789, 2021.

[16] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2021, no. 4, pp. 114–148, 2021.

[17] A. Al Badawi, B. Veeravalli, K. M. M. Aung, and B. Hamadicharef, "Accelerating subset sum and lattice based public-key cryptosystems with multi-core CPUs and GPUs," *J. Parallel Distrib. Comput.*, vol. 119, pp. 179–190, 2018.

[18] Z. Wang et al., "HE-Booster: An efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1067–1081, Apr. 2023.

[19] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A CUDA-accelerated word-wise homomorphic encryption library," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 5, pp. 4895–4906, Sep./Oct. 2024.

[20] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization*, 2020, pp. 264–275.

[21] J.-Z. Goey, W.-K. Lee, B.-M. Goi, and W.-S. Yap, "Accelerating number theoretic transform in GPU platform for fully homomorphic encryption," *J. Supercomputing*, vol. 77, pp. 1455–1474, 2021.

[22] O. Ozerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," *J. Supercomput.*, vol. 78, no. 2, pp. 2840–2872, 2022.

[23] A. Al Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "PrivFT: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226544–226556, 2020.

[24] L. Gao, F. Zheng, N. Emmart, J. Dong, J. Lin, and C. Weems, "DPF-ECC: Accelerating elliptic curve cryptography with floating-point computing power of GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 494–504.

[25] L. Gao et al., "DPF-ECC: A framework for efficient ECC with double precision floating-point computing power," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 3988–4002, 2021.

[26] R. Wei et al., "Heterogeneous-PAKE: Bridging the gap between PAKE protocols and their real-world deployment," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2021, pp. 76–90.

[27] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.

[28] X. Li, I. Laguna, B. Fang, K. Swirydowicz, A. Li, and G. Gopalakrishnan, "Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA GPUs," in *Proc. 32nd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2023, pp. 59–71.

[29] C. Yu, T. Chen, Z. Gan, and J. Fan, "Boost vision transformer with GPU-friendly sparsity and quantization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2023, pp. 22658–22668.

[30] P. Kluska, A. Castelló, F. Scheidegger, A. C. I. Malossi, and E. S. Quintana-Ort í, "QAttn: Efficient GPU kernels for mixed-precision vision transformers," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2024, pp. 3648–3657.

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

[32] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Proc. Int. Conf. Sel. Areas Cryptogr.*, Springer, 2018, pp. 347–368.

[33] F. Winkler, *Polynomial Algorithms in Computer Algebra*. Berlin, Germany: Springer Science & Business Media, 1996.

[34] R. Crandall and B. Fagin, "Discrete weighted transforms and large-integer arithmetic," *Math. Comput.*, vol. 62, no. 205, pp. 305–324, 1994.

[35] D. H. Bailey, "FFTs in external or hierarchical memory," *J. Supercomputing*, vol. 4, no. 1, pp. 23–35, 1990.

[36] IEEE Standards Committee and other, "IEEE standard for floating-point arithmetic," *IEEE Std*, vol. 754, no. 2008, pp. 1–70, 2008.

[37] J. Liu, G. Xiao, F. Wu, X. Liao, and K. Li, "AAPP: An accelerative and adaptive path planner for robots on GPU," *IEEE Trans. Comput.*, vol. 72, no. 8, pp. 2336–2349, Aug. 2023.

[38] L. Wan et al., "A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator," in *Proc. 27th Eur. Symp. Res. Comput. Secur.*, Copenhagen, Denmark, Springer, 2022, pp. 514–534.

[39] A. Al Badawi, B. Veeravalli, and K. M. M. Aung, "Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution," in *Proc. Future Inf. Commun. Conf.*, Springer, 2018, pp. 666–682.

[40] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, pp. 941–956, Second Quarter 2021.

[41] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Conf. Theory Appl. Cryptographic Techn.*, Springer, 1986, pp. 311–323.

[42] V. Shoup, "NTL: A library for doing number theory," 2021. Accessed: Aug. 2024. [Online]. Available: https://libntl.org/

[43] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[44] CUDA NVIDIA C Programming Guide 7.27.2, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#copy-and-compute-pattern-staging-data-through-shared-memory

[45] N. Emmart and C. C. Weems, "High precision integer multiplication with a GPU using strassen's algorithm with multiple FFT sizes," *Parallel Process. Lett.*, vol. 21, no. 03, pp. 359–375, 2011.

[46] H. L. Garner, "The residue number system," in *Proc. Papers Presented March 3-5, 1959, Western Joint Comput. Conf.*, 1959, pp. 146–153.

[47] H. Zhao et al., "Tacker: Tensor-CUDA core kernel fusion for improving the GPU utilization while ensuring QoS," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2022, pp. 800–813.

[48] H. Wang, W. Yang, R. Hu, R. Ouyang, K. Li, and K. Li, "A novel parallel algorithm for sparse tensor matrix chain multiplication via TCU-acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 8, pp. 2419–2432, Aug. 2023.

[49] Intel, “Intel homomorphic encryption (HE) acceleration library,” 2022. Accessed: Aug. 2024. [Online]. Available: https://github.com/intel/hexl

[50] cuHE, 2017. Accessed: Aug. 2024. [Online]. Available: https://github.com/vernamlab/cuHE

[51] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, “High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA,” *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2018, no. 2, pp. 70–95, 2018.

[52] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, “TensorFHE: Achieving practical computation on encrypted data using GPGPU,” in *Proc. 2023 IEEE Int. Symp. High- Perform. Comput. Archit.*, 2023, pp. 922–934.

[53] CKKS-GPU-CORE, 2021. Accessed: Aug. 2024. [Online]. Available: https://github.com/scale-snu/ckks-gpu-core

[54] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, “CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy,” in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2016, pp. 201–210.

[55] H. Chen, R. Cammarota, F. Valencia, F. Regazzoni, and F. Koushanfar, “AHEC: End-to-end compiler framework for privacy-preserving machine learning acceleration,” in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.

[56] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, “Low latency privacy preserving inference,” in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2019, pp. 812–821.

[57] Y. Zhu, X. Wang, L. Ju, and S. Guo, “FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference,” in *Proc. IEEE Int. Symp. High- Perform. Comput. Archit.*, 2023, pp. 896–907.

[58] A. Al Badawi et al., “Towards the AlexNet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs,” *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1330–1343, Third Quarter 2021.

[59] Y. LeCun et al., “MNIST handwritten digit database,” 2010. Accessed: Aug. 2023. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[60] G. Fan et al., “Towards faster fully homomorphic encryption implementation with integer and floating-point computing power of GPUs,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2023, pp. 798–808.

Fangyu Zheng received the BE degree in information security from the University of Science and Technology of China, Hefei, China, in 2011, and the PhD degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2016. He is currently an associate professor with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include applied cryptography and high-performance computing.

Guang Fan received the BE degree from the Beijing Institute of Technology, Beijing, China, in 2018, and the PhD degree in cyberspace security from the University of Chinese Academy of Science, Beijing, China, in 2023. He is currently an assistant researcher with Ant Group. His research interests include privacy-preserving computing and high-performance computing.

Wenxu Tang received the BE degree in information security from the University of Science and Technology of China, Hefei, China, in 2022. His research interests include applied cryptography and high-performance computing.

Yixuan Song received the EE degree from Shanghai Jiao Tong University, Shanghai, China, in 2022. He is currently an engineer with Ant Group. His research interests include privacy-preserving computing and high-performance computing.

Tian Zhou received the BE degree from Hangzhou Dianzi University. He is currently working toward the PhD degree with the University of Science and Technology of China. His research interest includes applied cryptography.

Yuan Zhao received the MS degree in software engineering from Peking University, Beijing, China, in 2013, and the PhD degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2016. He is currently working as an algorithm engineer with Ant Group. His research interests include secure multi-party computation, fully homomorphic encryption, and trusted execution environments.

Jiankuo Dong received the BE degree from Xi’an Jiaotong University, and the PhD degree from the University of Chinese Academy of Sciences, in 2014 and 2019, respectively. He is currently an associate professor with the School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, public key cryptography and applied cryptography.

Jingqiang Lin (Senior Member, IEEE) received the MS and PhD degrees from the University of Chinese Academy of Sciences, in 2004 and 2009, respectively. His research interests include applied cryptography and system security.

Shoumeng Yan is the CTO of Ant Misuan, an Ant Group company, and the head of computing systems lab of Ant Research. He leads the development of a series of high-profile open-source projects including Occlum TEE OS, HyperEnclave TEE, and Asterinas OS. His current research interests focus on secure and trustworthy computing systems.

Jiwu Jing (Member, IEEE) received the BE degree from the Department of Electronics Engineering, Tsinghua University, Beijing, China, in 1987, and the MSc and PhD degrees from the Graduate School, Chinese Academy of Sciences, Beijing, in 1990 and 2003, respectively. He is currently a professor with the School of Cryptology, University of Chinese Academy of Sciences. His main research interest is information security, including public key infrastructure, identity management, and fault tolerance.