2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00085

Towards Faster Fully Homomorphic Encryption Implementation with Integer and Floating-point Computing Power of GPUs

Guang Fan[*][†][§], Fangyu Zheng[*][‡][§], Lipeng Wan[*][†][§], Lili Gao[¶], Yuan Zhao[‖], Jiankuo Dong[**], Yixuan Song[‖], Yuewu Wang[‡], Jingqiang Lin[††]

[*]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

[†]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

[‡]School of Cryptography, University of Chinese Academy of Sciences, Beijing, China

[§]Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China

[¶]Department of Computer and Software, Nanjing University of Information Science and Technology, Nanjing, China

[‖]Ant Group, Hangzhou, China

[**]School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China

[††]School of Cyber Security, University of Science and Technology of China, Hefei, China

***Abstract*—Fully Homomorphic Encryption (FHE) allows computations on encrypted data without knowledge of the plaintext message and currently has been the focus of both academia and industry. However, the performance issue hinders its large-scale application, highlighting the urgent requirements of high-performance FHE implementations.**

With noticing the tremendous potential of GPUs in the field of cryptographic acceleration, this paper comprehensively investigates how to convert the available computing resources residing in GPUs into FHE workhorses, and implement a full set of low-level and middle-level FHE primitives based on two arithmetic units (i.e., INT32 and FP64 units) with three types of data precision (i.e., INT32, INT64 and FP64). This paper gives a comprehensive evaluation and comparison based on each roadmap. Our implementations of fundamental functions outperform the implementations on the same platform by $1.7\times$ to $16.7\times$. Taking CKKS FHE schemes as a case study, our implementation of homomorphic multiplication achieves $3.2\times$ speedup over the state-of-the-art GPU-based implementation, even considering the difference of platforms. The detailed evaluation and comparison of this paper would offer a vital reference for the follow-up work to choose appropriate underlying arithmetic units and important primitive optimizations in GPU-based FHE implementations.

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 of 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 great algorithmic advances, considerable performance overheads are still a bottleneck that restricts further employment of FHE. The urgent requirement for FHE accelerations has been highlighted and attracted wide attention.

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 [10] and HElib [11], 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.

Fortunately, the rapid evolution of graphics processing units (GPUs) brings an opportunity to solve the performance dilemma of FHE. Single instruction, multiple threads (SIMT)

This work was partially supported by National Natural Science Foundation of China under Award No. 61902392, the Natural Science Foundation of Jiangsu Province under Grant BK20220388, and CCF-Huawei Populus Grove Fund. *(Corresponding author: Fangyu Zheng, E-mail: zhengfangyu@iie.ac.cn.)*

introduced by NVIDIA in 2006 is an execution model that is used in GPUs for parallel computing. SIMT utilizes thread-level parallelism, i.e., multiple independent threads execute concurrently using a single instruction. This feature brings tremendous potential to FHE acceleration, which requires intensive arithmetic operations. Many previous works attempted to exploit this tremendous arithmetic computing power and memory bandwidth. Previous studies [14], [15], [16], [17] have leveraged GPU to accelerate the fundamental operations (e.g., CRT and NTT) of FHE. And some previous works [18], [19], [20] dedicate to exploring efficient NTT implementations on GPU for FHE. Based on the GPU FHE implementation, Al Badawi et al. [21] present a text classification solution, PrivFT, using FHE to preserve the privacy of content.

By implanting and parallelizing the conventional CPU implementations, the performance of the GPU-based implementations improves by tens to hundreds of times compared with that of CPU. However, they fall short in adapting the FHE workload well into the instruction set, memory hierarchy and architecture of GPUs. CPU and GPU orient different types of workloads and thus follows different design principle. From the perspective of FHE implementation, the selection of instruction set and the design of algorithm structure need to be reconsidered when implementing a GPU-based FHE.

Taking the underlying instruction set as an example, both high-definition 3D graphics processing and deep learning applications require high-speed floating-point processing capabilities. In GPUs, floating-point computing power naturally outperforms the integer one. From 2010 to the present, the floating-point computing power of NVIDIA GPUs has grown over ten-fold, from 1.345 (Fermi architecture) Tera Floating-point Operations Per Second (TFLOPS) to 40 (Ampere architecture) TFLOPS, while some generations of GPUs (e.g., Maxwell and Pascal architecture) lack of full-function integer units. With noticing the great potential of the floating-point computing power, prior studies [22], [23], [24] reported up to $3\times$ speedup for RSA and ECC with floating-point computing power of GPUs compared to that based on integer one. However, there is no research on the application of floating-point computing power of GPUs to FHE implementation.Besides, the implementation of CRT and NTT algorithms used in FHE implementations are worth further exploring. On the one hand, the adaptation of different GPUs, different underlying arithmetics and different implementation methods needs to be obtained through experiments. On the other hand, the algorithms, especially the NTT algorithm, have room for further optimization.

In this paper, we try to implement a faster FHE implementation on GPUs with fine-grained optimization and multiple computing units, including INT32 units and FP64 units. Our contributions are three-fold:

- Firstly, we investigate all the available computing resources residing in GPUs and give a detailed analysis of how to turn these resources into FHE workhorses. Then, we implement a full set of underlying arithmetic required by FHE schemes with three types of data precision to fully leverage the INT32 units and FP64 units, especially the FP64-based implementation, which is a pioneering work to employ floating-point arithmetic in FHE accelerations.
- Secondly, based on all the three series of the underlying arithmetic, we develop all the fundamental middle-level primitives including CRT/ICRT and NTT/INTT. We decouple their main computational load, and exploit various optimization algorithms to achieve various implementations. Pre-calculation and GPU memory optimizations are also adopted. For NTT, we design and apply different negacyclic convolution strategies on the basis of a recursive 4-step NTT layout, including the fusion of pre-processing and post-processing into the inner NTT.
- Finally, we give a comprehensive evaluation and comparison on fundamental middle-level primitives with different underlying arithmetics and different optimization algorithms. Our best FP64-based implementation achieves $1.7\times$ to $16.7\times$ speedup over the implementations of other works running on the same platform. Taking CKKS FHE schemes as a case study, our implementation of homomorphic multiplication on A100 GPU outperforms the state-of-the-art GPU-based implementation by .32$\times$.

The rest of our paper is organized as follows. Section II presents background material. Section III gives an overall architecture of the implementation. Section IV detail the implementation. Section V analyses the performance of proposed algorithm and compares it with previous works. Section VI concludes the paper.

II. Background

This section reviews the essential concepts that are the key to understanding the rest of the paper. This section starts with the notations used throughout the paper and then introduces the fundamental algorithms for efficient FHE implementation. Finally, we give a brief introduction to the numerical format of floating-point numbers.

A. Notation

For an integer q, we identify $\{0, 1, \cdots, q-1\}$ as a representative of $\mathbb{Z}_q$ and use $[z]_q$ to denote the reduction of the integer z, modulo q into that interval. $\boldsymbol{R}_q$ is the polynomial ring $\mathbb{Z}_q[x]/\{x^N+1\}$, where N is a power of 2. Ring arithmetic is performed modulo both $x^N + 1$ and q. A polynomial $\mathbf{a}(x) = \sum_{i=0}^{N-1} a_i \cdot x^i$ in $\boldsymbol{R}_q$ can also be represented as a vector over $\mathbb{Z}_q$, so that $\mathbf{a} = [a_0, a_1, \cdots, a_{N-1}]$. Similarly, we use $\mathbf{a}[i]$ to represent the coefficient of polynomial $\mathbf{a}(x)$ at position i.

Throughout the paper, q and N represent the coefficient modulus and the degree of the polynomial ring, respectively. β represents the word size, which refers to the number of bits processed by an multiplication instruction (or simulated multiplication instruction).

B. FHE Workloads

Many popular homomorphic encryption schemes are based on ring learning-with-error (RLWE) problem, including

BGV [3], BFV [25] and CKKS [5]. The polynomial arithmetic under cryptographic primitives is the main load of those schemes, especially the polynomial multiplications. The Chinese Remainder Theorem (CRT) and Number Theoretic Transform (NTT) are widely used in HE libraries to reduce the complexity of polynomial arithmetic.

1) Chinese Remainder Theorem: The Chinese Remainder Theorem (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 $\{a_0, a_1, \cdots, a_{r-1}\}$, 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 $O(l^2)$ to $O(r)$ for $l = \lceil \log q / \log \beta \rceil$ ($r \ll l^2$), but also can be processed in parallel.

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

$$a = \sum_{i=0}^{r-1} \frac{q}{q_i} \cdot ((\frac{q}{q_i})^{-1} \cdot a_i \mod q_i) \mod q \tag{1}$$

2) Number Theoretic Transform: Number Theoretic Transform (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 normal polynomial representation to NTT 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: $A_k = \sum_{j=0}^{N-1} a_j \omega_N^{jk} \mod q_i$ for $k \in \{0 \le k < N\}$, ω_N is the primitive N-th root of unity of NTT for $\mathbb{Z}_{q_i}$ ($\omega_N^{jk} \mod q_i$ 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 [26] is involved to avoid those operations. The process is shown in Equation (2), in which $\mathbf{\Omega^{-1}} = (1, \omega_{2N}^{-1}, \omega_{2N}^{-2}, \ldots, \omega_{2N}^{1-N})$.

$$\mathbf{c} = \mathbf{\Omega^{-1}} \odot INTT_N(NTT_N(\bar{\mathbf{a}}) \cdot NTT_N(\bar{\mathbf{b}})) \tag{2}$$

$\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$ are the vectors of the coefficients of $\mathbf{a}(\omega_{2N}x)$, $\mathbf{b}(\omega_{2N}x)$ respectively. Consider $NTT(\bar{\mathbf{a}}) = \mathbf{A}$, $A_k = \sum_{j=0}^{N-1}(a_j\omega_{2N}^j)\omega_N^{jk}$. To reduce the number of computations, it is usually simplified to $A_k = \sum_{j=0}^{N-1} a_j \omega_{2N}^{j(2k+1)}$. The transform of $NTT(\bar{\mathbf{a}})$ is also called Discrete Weighted Transform (DWT) [27]. Similarly, $INTT(\mathbf{A} \cdot \mathbf{\Omega^{-1}})$ is the inverse of DWT (IDWT).

Compared with traditional iterative NTT, four-step NTT [28] 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 Equation (3). In which, $j = j_1 + j_2 N_1$, $k = k_2 + k_1 N_2$.

$$A_{(k_2,k_1)} = \prod_{j_1=0}^{N_1-1} \prod_{j_2=0}^{N_2-1} a_{(j_1,j_2)} \omega_{N_2}^{j_2k_2} \omega_{N_1N_2}^{j_1k_2} \omega_{N_1}^{j_1k_1} mod\ q_i. \tag{3}$$

The above equation reveals the new four steps of the algorithm: (1) proceed N_1 groups of N_2-point NTT; (2) transpose the matrix; (3) multiply twiddle factors; (4) proceed N_2 groups of N_1-point NTT.

C. Floating-Point Numbers and the FMA Operation

The 32-bit and 64-bit basic binary floating-point formats defined in IEEE 754 [29] 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 [29], 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. System Analysis and Design

This section gives a detailed analysis of how to turn GPUs' resources into FHE workhorses and gives an overall design of our implementation.

A. FHE Workloads Analysis

The implementations defined over polynomial rings can be divided into three levels, homomorphic procedures, polynomial arithmetic, and underlying arithmetic.

- The high layer is composed of cryptographic primitives like key generation, encryption and decryption, and evaluation procedures like homomorphic multiplication and homomorphic addition.

- The middle layer is polynomial arithmetic, including polynomial addition, polynomial multiplication and representation domain transformation algorithms, like CRT and NTT.
- Under above all are the multi-word arithmetic and finite field arithmetic.

In FHE implementations, on account of the large scale of polynomial parameters (e.g., $q = 2^{1200}$, $N = 2^{16}$), the overhead of polynomial arithmetic is significant. Following the so-called "double-CRT" representation techniques [30], most polynomial arithmetic, especially polynomial multiplications, are performed in NTT domain to lower computational complexity and increase parallelism. CRT and NTT are required to transform the representation of polynomials from polynomial domain to NTT domain.

Although NTT and CRT significantly improve the performance of polynomial multiplication, NTT and CRT themselves are still very time-consuming. In HEAAN [31], a library of CKKS, a homomorphic multiplication takes 3,903 ms, among which CRT, NTT and their inverse operations account for 95.8% [15]. Therefore, improving the performance of CRT and NTT is an important aspect of FHE acceleration.

GPUs can significantly benefit large-scale computing, but would also make the implementation more complicated. There are many factors to consider, including arithmetic logic unit selection, parallel strategy, inter-thread communication, multi-level memory selection, etc. These designs are essentially important for the FHE schemes with large-scale parameters.

B. GPU Native Instruction Set Selection

There are many computing resources in modern GPUs. For example, there are four types of arithmetic logic units (ALUs) that reside in every streaming multiprocessor (SM) of the latest Tesla A100 GPU, including integer (INT32) cores, two types of floating point cores (FP32 and FP64), and AI-dedicated tensor cores. These resources support GPUs for different types of computations. Among them, the throughput of FP32 arithmetic instructions is no less than that of INT32. NVIDIA GPUs also support native 64-bit floating-point arithmetic but not native 64-bit integer arithmetic. On some generations of GPUs, since there is no full-function integer unit, integer multiplications are far less efficient than floating-point multiplications. The instruction throughput of the Tesla A100 and P100 GPUs is shown in Table I, where the integer multiplication on P100 requires multiple instructions to complete. When performing calculations, INT32 multiplication is more than twice as slow as FP32 multiplication. Taken together, GPU invests more resources in floating point numbers.

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.

TABLE I: Throughput of native integer and floating-point arithmetic instructions on A100 and P100 GPU (number of operations per clock cycle per SM)

Throughput(/SM/CLK)	SPF	DPF	INT32 Add	INT32 Mul
A100	64	32	64	64
P100	64	32	64	*

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

With noticing that in the GPU, there are two types of floating point computation units, FP32 units and FP64 units, whose significant bits are 24-bit and 53-bit respectively. In the implementation of FHE, CRT and NTT algorithms usually need to choose a large modulus, generally 30-bit or above. The use of FP32 units will limit the choice of modulus size, and the proportion of sigficand in FP32 to the total length is also relatively low. In addition to integer units and floating point units, the latest GPUs also include tensor cores. Some studies use tensor cores for cryptographic computations, such as lattice-based cryptography [32]. However, tensor cores mainly support low word-size operations, such as INT8, FP16, and thus fit well with the lattice-based cryptography whose underlying modulus is relatively small but are highly inefficient for the FHE calculation.

For the above considerations, we choose INT32 units and FP64 units. In fact, from the perspective of the final implementation, these two computing units have their own advantages and disadvantages.

C. Overall Design

In order to achieve faster FHE implementation, we implement a full set of low-level and middle-level FHE primitives with comprehensive optimizations. Our work focus on the common used polynomial ring $\mathbb{Z}_q[x]/\{x^N + 1\}$. The overall design is shown as Fig. 1.

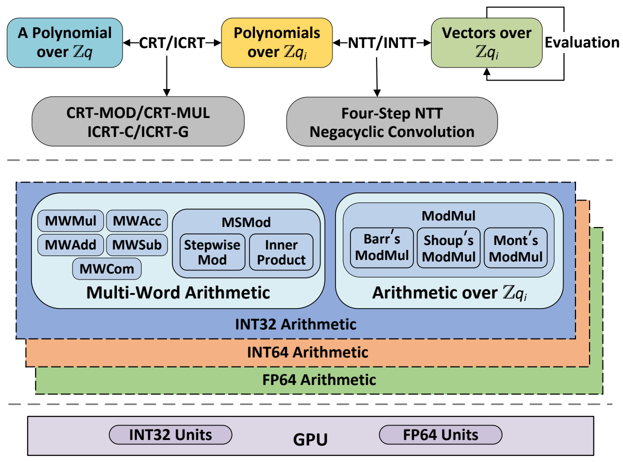


Fig. 1: The overall design

1) Underlying arithmetic: There were studies on the implementation of FHE that leveraged two kinds of integer arithmetic respectively. Previous studies [14], [33], [15] used 32-bit word size, while other studies [18], [16], [34] chose 64-bit word size. Because different researches use different optimization algorithms, it is difficult to directly compare which underlying arithmetic implementation is better. Besides, there is no research to demonstrate an FHE implementation using FP64 arithmetic.

Thus, we select **INT32**, **INT64** and **FP64 arithmetic** as the three candidates and implement all the three kinds of underlying arithmetic, trying to explore the best routine for the FHE implementations. Their word size are 32-bit, 64-bit and 52-bit respectively. The first two are based on INT32 units of GPUs and the last one is based on FP64 units.

Two sets of components are involved in this level. Multi-word arithmetic is mainly used in CRT/ICRT algorithm. Multi-word accumulations (MWAcc) and multi-word multiplications (MWMul) are most time-consuming operations. Multi-word integer modulo single-word integer (MSMod) is the main computing load in CRT and also appears in the ICRT. We implement that operation in 2 ways, using step-wise modular reductions and an inner product of two sequences respectively. Finite field arithmetic over $\mathbb{Z}_{q_i}$ is mainly used in NTT/INTT algorithm and homomorphic evaluations. In those operations, modular multiplication (ModMul) is one of the performance bottlenecks. We use Barrett's ModMul [35], Shoup's ModMul [36] and Montgomery's ModMul [37] to accelerate those operations.

2) Polynomial arithmetic: The polynomial arithmetic is built upon the underlying arithmetic. Firstly, we determine the choice of moduli used in CRT and NTT. Generally, there are two options : 1) using the same single-word primes for both CRT and NTT and make $q_i = 1 \mod 2N$ to satisfy the condition of negacyclic convolution; 2) using a special modulus for NTT to improve the computing performance. Previous works [38], [14] introduced Solinas prime, $q_s = 2^{64} - 2^{32} + 1$, to convert modular reductions into shift, addition and subtraction operations. Nevertheless, the method also introduces many limitations including the contradiction with negacyclic convolution. Exploiting the fast modular reduction algorithm and releasing both floating-point and integer computing power of GPUs, the special prime is no longer needed, so we choose same moduli with CRT for NTT.

Based on the different word size of INT32, INT64 and FP64 arithmetic, we choose primes under 29-bit, 61-bit and 49-bit, respectively, reserving 3-bit for lazy reduction.

There are some components or methods that can be substituted for each other in our system. Nevertheless, each method or component involves different kinds of computation load, and the required memory size also differs. The performance may correspondingly differs for different underlying arithmetic and different generations of GPUs. Therefore, we do not employ one certain method but explore and evaluate different technical routes in our implementations.

For CRT, We explore multiple optimizations of MSMod and develop two ways of CRT, namely CRT-MOD and CRT-MUL. We also investigate two ways for ICRT, namely classic ICRT (ICRT-C) and Garner's algorithm (ICRT-G). Their main computational loads are decoupled and optimized with our underlying arithmetic. For NTT, we design different negacyclic convolution strategies on the basis of a recursive 4-step NTT layout, including the fusion of pre-processing and post-processing into the inner NTT. The adaptability of different ModMul algorithms to NTT with a large degree ($2^{14} \sim 2^{16}$) is also studied.

3) Parallel Strategy: For GPU-based FHE implementations, one straightforward way is to process an instance in a single thread, which would avoid the communication overhead between threads and have high throughput when processing a large number of instances. The other way is to process an instance by multiple threads, which is able to make full use of multi-level memory resources and shorten the delay of processing a single instance.

Since the parameters of FHE schemes are large in scale, the single-threaded method would cause unacceptable high latency. Besides, unlike CPU, the register files are assigned to each GPU thread, and each thread occupies a large number (up to 255) of dedicated 32-bit resisters, and an important lesson learned from our previous GPU-based implementations is to place the frequently-used variables/parameters into the abundant register resources as much as possible, however, even such a large register number cannot afford the large-scale FHE parameters. Based on the above considerations, we use multiple threads to collaborate on one polynomial operation.

4) Memory Usage: Compared with CPU, the memory hierarchy of GPUs is more complicated. Roughly, the access speed of each memory is: *register* $>$ on-chip memory (e.g., *shared memory* and L1 cache) $>$ off-chip cached memory (e.g., *constant memory* and *texture memory*) $>$ off-chip memory (e.g., *global memory* and *local memory*). And we deal with different types of data in FHE computation process as follows:

- The input and output data of CRT and NTT algorithm are of quite large sizes, so we have put them in the *global memory*.
- Precomputed data, including moduli and values derived from moduli and β, which are small and read-only, are placed in the read-only cached *constant memory*.
- Other precomputed data, like twiddle factors in NTT, are too numerous to be allocated in constant memory. In INT32 arithmetic, we store them in the read-only cached *texture memory*. While, in INT64/FP64 arithmetic, they are just placed in global memory, because the texture memory is designed for 32-bit memory access and has the opposite effect for 64-bit one.
- As aforementioned, the FHE workloads are highly parallelized so that the intermediate data generated by threads can be stored in *registers* as much as possible to fully leverage the potential of ALUs. When inter-thread data exchange is required, we use *shared memory* for communication within the block first and then use *global memory* for communication between blocks. The data is arranged elaborately to maximize coalesced global memory access and minimize bank conflicts of shared memory.

IV. Implementations

In this section, we first introduce the underlying arithmetic including instruction-level arithmetic, multi-word arithmetic and finite field arithmetic. The underlying arithmetic is implemented in three versions, using INT32 arithmetic, INT64 arithmetic and FP64 arithmetic respectively. Then detailed implementation of the CRT/ICRT and NTT/INTT is also further explained.

A. Instruction-level Arithmetic

INT32 arithmetic. NVIDIA GPUs support 32-bit native integer instructions. In CUDA parallel thread execution (PTX) instruction set, multiply-add (`mad`) instruction and instructions for carry operations (`addc`, `subc` and `madc`) could be directly utilized to improve efficiency. In order to apply those instructions, we directly adopt PTX assembly to exploit the native INT32 instructions in INT32 arithmetic.

INT64 arithmetic. Since there is no INT64 computing units in NVIDIA GPUs, INT64 operations need to be simulated with INT32 instructions. To handle single-word multiplications, we use the built-in functions `__umul64hi` and `__umul64lo` in CUDA Math API. An INT64 multiplication is substituted by four INT32 multiplication instructions and six addition instructions in total. For carry operations in INT64 multi-word operations, we leverage the native INT32 instructions and the carry flag `cc`, which will be detailed in Section IV-B1.

FP64 arithmetic. In FP64 arithmetic, we combine the advantages of floating-point computing power and integer computing power of GPUs, so that both multiplication and addition get the best performance. For multiplication, we use FP64 instruction, which is twice as fast as INT64 multiplication in some GPUs and is not slow in the rest.

Algorithm 1 demonstrate the single-word multiplication (SWMul) in FP64 arithmetic. By exploiting all fraction part of FP64 numbers, the word size of FP64 Arithmetic reaches 52-bit. We utilize two `fma` instructions to calculate the high 52-bit and the low 52-bit of the product of single-word multiplication 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 fhi, 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 fhi will not be affected by the low 52-bit product. Next, we subtract the high 52-bit from the product, and fix the the most significant bit of flo by setting the *anchor* value to 2^{52}, which is equivalent to setting the third operand of the `fma` instruction to $(2^{52} - (fhi - 2^{104}))$. 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 that does not require further operations.

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: $\mathbf{mov.d64} \quad h, f\_hi$
6: $\mathbf{mov.d64} \quad l, f\_lo$
7: $h = h \ \& \ mask$
8: $l = l \ \& \ mask$
```

In FP64 arithmetic, the integer computing power is also exploited to handle additions and subtractions for two reasons. Firstly, there is no FP64 instruction to efficiently resolve carry operations. Secondly, 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.

B. Underlying Arithmetic

1) Multi-Word Arithmetic: Multi-Word arithmetic is mainly used in CRT/ICRT algorithm, where multi-precision representation is used for multi-word coefficients.

The multi-word accumulation (MWAcc) is a tedious operation for repeated 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.

Multi-word multiplication-accumulation (MWMAC), which is commonly used in the ICRT algorithm, 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 main computing load of CRT algorithm, which will be illustrated in Section IV-C.

2) Finite Field Arithmetic over $\mathbb{Z}_{q_i}$: Finite Field Arithmetic includes modular addition (ModAdd), modular subtraction (ModSub) and modular multiplication (ModMul). The ModMul is the most time-consuming one. To mitigate the computational overhead, three ModMul algorithms are implemented, aiming at different situations.

When a multiplier in ModMul can be determined in advance, Montgomery's ModMul [37] and Shoup's ModMul [36] are better options. The difference is that the former needs one more half-multiplications (4) than the latter (3), but requires fewer inputs. When the multiplier is runtime generated, Barrett's ModMul [35] is the choice.

Note that the lower word of the multiplication product cannot be calculated separately with the FMA instruction. So in FP64 arithmetic, we use the `__umul64lo` function to compute the single lower word of the product.

C. The Chinese Remainder Theorem and its Reconstruction

1) CRT: In the CRT algorithm, calculating the remainders of the coefficients is the main computing load. A multi-word integer (MWInt) a is represented by multi-precision representation, i.e., $(a_0, ..., a_{l-1})$, such that $a = \sum_{j=0}^{l-1} a_j \beta^j$. 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, refer to as CRT-MOD and CRT-MUL respectively.

CRT-MOD. Previous work [14] used a stepwise-modulo method to implement the MSMod operation. Let $h_0 = a_0$. This method calculates $h_{j+1} = h_j \cdot \beta + a_{j+1} \mod q_i$ for $j \in \{0 \leq j < l-1\}$ iteratively. However, the native modulo operation (%) in GPUs is very time-consuming. We use the Barrett reduction [35] to accelerate the modulo operations. There is no way to use Shoup's modular reduction here, because both multipliers are not determined until runtime. Besides, the precomputed data $ratio'_i = \beta^2/q_i$ instead of $ratio_i = 2^{\lceil 2\log q_i \rceil}/q_i$ are used here since the input number is longer than the input of Barrett's ModMul in arithmetic over $\mathbb{F}_{q_i}$.

CRT-MUL. The other approach to computing MSMod is based on the idea of precomputation. The β and q_i are independent of the input, thus we can precompute $\beta^j \mod q_i$ for all j as sequence $\mathbf{b}$. In this way, an MSMod is transformed into the inner product of two sequences: $a \mod q_i = \sum_{j=0}^{l-1} a_k \cdot \beta^j \mod q_i = \sum_{j=0}^{l-1} a_j \cdot (\beta^j \mod q_i) \mod q_i = \mathbf{a} \odot \mathbf{b} \mod q_i$. In our implementation, the inner product is performed by a batch of single-word multiplications 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 [39] 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 Equation (1) indicates. The main computation loads are MWMAC and multi-word modulo (MWMod) operations. The MWMAC is analyzed in Section IV-B2, 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 $q_i, q, ratio_i, q/q_i$ and $(q/q_i)^{-1} \mod q_i$ for the classic ICRT algorithm and store them in constant memory.

Algorithm 2 ICRT Using Garner's Algorithm

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

ICRT-G. Another CRT reconstruction approach, adopting Garner's algorithm [39], is shown in Algorithm 2. We precompute $k_i = \prod_{j=0}^{i-1} q_j$ and $c_i = \prod_{j=0}^{i-1} (q_j^{-1} \mod q_i) \mod q_i$ for $i \in \{1 \leq i < r\}$ and store them in constant memory. ICRT-G has different computational loads from ICRT-C. It also contains MWMAC operations in line 14, but the average word number of k_i is obviously shorter than that of q/q_i. Besides, there are MSMod operations in line 12, for which we employ the method used in CRT-MUL implementation.

D. Number Theoretic Transform and its Inverse

1) NTT Layout: To improve the available multiplication times under a certain level of security, HE schemes usually use a large N, such as 2^{15}, which also increases the computational cost, especially for NTT. To allow multiple SMs to cooperate in NTT and minimize latency, four-step NTT is adopted.

The NTT layout is shown as Fig. 2. We exploit four-step NTT recursively, and make $N = (N_3 \times N_2) \times N_1 = (64 \times 64) \times \frac{N}{4096}$, $N \in \{2^{14}, 2^{15}, 2^{16}\}$. The input coefficients are regarded as a three-dimensional array, with N_1 as the main order and N_2 as the secondary order.

The NTT function is divided into three parts and consists of seven steps: (i) proceed N_2 groups of N_3-point NTT; (ii) multiply twiddle factors of $(N_2 \times N_3)$-point NTT; (iii) transpose the matrix; (iv) proceed N_3 groups of N_2-point NTT. (v) multiply twiddle factors of (N)-point NTT; (vi) proceed $(N_2 \times N_3)$ groups of N_1-point NTT; (vii) transpose the matrix.

In order to synchronize data between threads, we use three kernels to handle three parts respectively. Threads access the memory at the beginning of each kernel and write back to the memory at the end of each kernel. The transpose is merged into data writing. A line of coefficients in the main order is always accessed by adjacent threads to maximize merged memory access.

The naïve NTT algorithm requires at least m memory accesses to the data set to compute a 2^m-point NTT, which is the bottleneck of NTT implementations on GPU. In our implementation, the number of global memory accesses is reduced to three times. It takes merged memory access into consideration, otherwise the number of kernels can be even less.

For the inner NTT in step one, four and six, shared memory and registers are utilized. The 64-point NTT in kernel one and two are further decomposed into (8×8)-point NTT by four-step NTT method. The basic component in each kernel is 4/8-point NTT, processed by a single thread. We use shared memory to exchange data between threads in a block.

2) Negacyclic Convolution Strategy: As the primes we select satisfy $q_i \equiv 1 \mod 2N$, negacyclic convolution can be introduced into our implementation to avoid doubling the NTT length and zero-padding.

In addition to adopting pre-processing and post-processing of negacyclic convolution, we also have another implementation, merging them into four-step NTT as Equation (4), in which, $j = j_2 + j_1 N_2$, $k = k_1 + k_2 N_1$, to reduce memory accesses and modular multiplications.

$$A_{(k_2,k_1)} = \prod_{j_1=0}^{N_1-1} \prod_{j_2=0}^{N_2-1} a_{(j_1,j_2)} \omega_{2N_2}^{j_2(2k_2+1)} \omega_{2N_1N_2}^{j_1(2k_2+1)} \omega_{N_1}^{j_1k_1}. \quad (4)$$

The above equation reveals the new four steps of the algorithm: (1) proceed N_1 groups of N_2-point DWT; (2) transpose the matrix; (3) multiply twiddle factors of N-point DWT; (4) proceed N_2 groups of N_1-point NTT.

After iteratively using the above algorithm, a round of inner NTT is converted into DWT, which becomes more complicated, but the overhead of memory access and ModMuls in explicit pre-processing and post-processing are eliminated.

Besides, this method makes it necessary to implement independent INTT. Originally, we only need to change the input order of NTT to get INTT. After merging, INTT becomes as shown in Equation (5). According to the equation, the four steps of the INTT algorithm are: (1) proceed N_1 groups of N_2-point INTT; (2) transpose the matrix; (3) Multiply twiddle factors of N-point IDWT; (4) proceed N_2 groups of N_1-point IDWT.

$$a_{(j_2,j_1)} = \frac{1}{N} \prod_{k_1=0}^{N_1-1} \prod_{k_2=0}^{N_2-1} A_{(k_1,k_2)} \omega_{N_2}^{-j_2k_2} \omega_{2N_1N_2}^{-j_2(2k_1+1)} \omega_{2N_1}^{-j_1(2k_1+1)}. \quad (5)$$

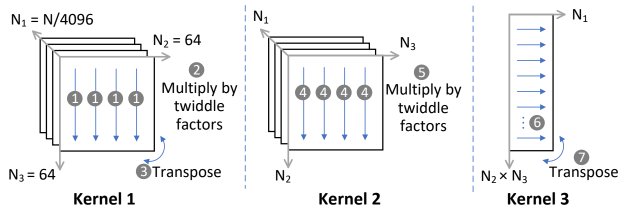


Fig. 2: NTT layout

3) ModMul Strategy: Modular multiplication is the performance bottleneck in NTT algorithm. There are a lot of ModMuls in Equation (4), and the internal 4/8-point NTT also contains ModMuls. One multiplier of these ModMuls is a twiddle factor, which can be obtained by pre-computation. In this case, we introduce Montgomery's or Shoup's ModMul algorithm in our implementation. Among them, Shoup's ModMul requires fewer multiplications, but it needs to store both twiddle factors and their variants, making the memory access pressure twice that of Montgomery's ModMul.

In addition, we carry out lazy reduction, which can greatly reduce the number of comparisons and modular subtractions. Three bits are reserved when we choose the size of the modulus, so the temporary value can reach $8q_i$ at most. We remove the last step of Montgomery's and Shoup's ModMul in NTT, allowing the ModMul result between 0 and $2q_i$. The inner NTT also becomes more concise. Reducing to q_i is performed before NTT is completed.

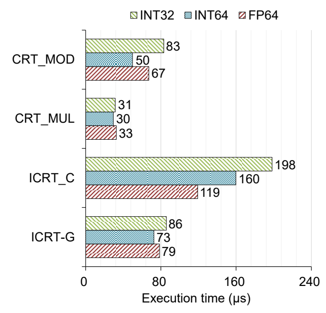
4) Thread Organization: For each kernel, $N/8$ threads are launched to handle a polynomial of degree N over $\mathbb{F}_{q_i}$, so that each thread can handle an eight-point NTT or two four-point NTT in the inner process. Considering the utilization of shared memory, 64 threads form a block and each block exploits 2k/4k/4k bytes of shared memory in the implementation utilizing INT32/INT64/FP64 arithmetic. Totally, three kernels can processes all r NTT of r polynomials over $\mathbb{F}_{q_i}$ for $i \in \{0 \leq i < r\}$ by using $\frac{r \cdot N}{512}$ blocks and each block has 64 threads.

V. Evaluation and Discussion

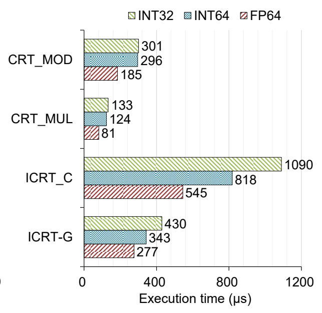
In this section, we evaluate the performance of CRT and NTT implementations with different underlying arithmetics and different optimization techniques. Besides, we integrate our low-level and middle-level FHE primitives into the open source library, HEAAN [31], implement the homomorphic multiplication, and compare the performance with the other CPU-based or GPU-based implementations.

A. Experimental Setup

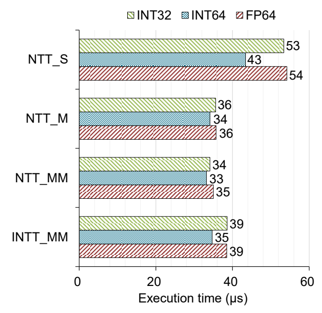
The experiments are carried out on two generations of NVIDIA GPUs, Tesla A100 and P100, which represent the GPUs with and without full-function integer cores respectively. Tesla P100 and A100 consist of 56 and 108 SMs performing up to 9.53 and 19.49 trillion FP32 float operations per second, and are operating at 1.30GHz and 1.38 GHz respectively.



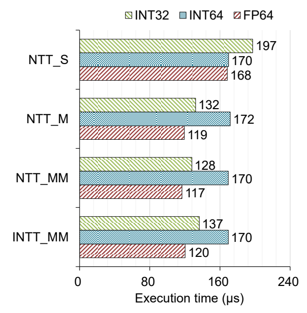
(a) CRT/ICRT Imp. on A100



(b) CRT/ICRT Imp. on P100



(c) NTT/INTT Imp. on A100



(d) NTT/INTT Imp. on P100

Fig. 3: Execution time of CRT and NTT implementations on A100 and Tesla P100

B. Evaluation of CRT and NTT

We evaluate CRT and NTT implementations with different optimization algorithms and three versions of underlying arithmetic respectively. In those experiments, the representative parameters are $(N = 2^{15}, \log q \approx 881)$. The number of moduli r is $31/15/18$ for INT32/INT64/FP64 arithmetic, respectively.

1) CRT evaluation: We compare two CRT implementations, CRT-MOD and CRT-MUL, and also two ICRT implementations, ICRT-C and ICRT-G. Fig. 3a and 3b shows 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.

ICRT is more time-consuming than CRT. Garner's algorithm improves the performance of ICRT significantly in almost all implementations. 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.

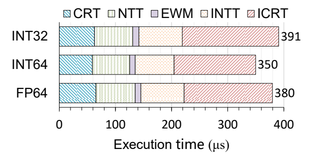
It can be noticed that FP64 implementation performs better than other implementations in ICRT implementations, which demonstrates the high efficiency of FP64 version MWSum and MWMul operations. Integer implementations of CRT-MUL have obvious advantages over FP64 implementations on A100 thanks to the utilization of MAD instructions.

2) NTT evaluation: For NTT, we compare three implementations with four-step NTT algorithm. The first two setups (NTT-S and NTT-M) exploit Shoup's ModMul and Montgomery's ModMul respectively. They both adopt negacyclic convolution with pre-processing and post-processing. The third one (NTT-MM) utilizes Montgomery's ModMul and also merges pre-processing and post-processing into four-step NTT.

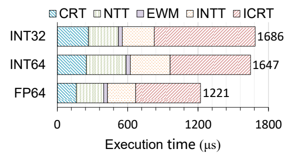
First, we compare the different modular reduction algorithms used in the first two setups. The result on A100 shows that Montgomery's ModMul is at least 21% faster than Shoup's ModMul in implementations with different arithmetics. This indicates that Montgomery's ModMul with fewer memory accesses is more suitable for NTT than Shoup's ModMul with fewer multiplication times. As to the implementations on P100, the results of the first two setups with INT64 arithmetic are almost the same, indicating that when the clock speed of GPU decreases, the two kinds of modular reduction algorithms with different characteristics represent similar performance. In addition, Montgomery's ModMul is at least 29% faster than Shoup's ModMul in both INT32 arithmetic and FP64 arithmetic, because, in them, a lower half multiplication needs to be obtained by processing a full multiplication.

Then, we compare different negacyclic convolution strategies in NTT-M and NTT-MM. It can be seen from the result that the influence of eliminating pre-processing and post-processing in NTT-MM is partially offset by the increased overhead of the inner DWT. The advantage of merging operations is not obvious.

The situation of INTT is basically the same as that of NTT with a slight increase in time consumption.



(a) Implementation on A100



(b) Implementation on P100

Fig. 4: Execution time of a polynomial multiplication on A100 and Tesla P100

3) Analysis of different arithmetics: To further compare the effects of different underlying arithmetics, we compare the results of polynomial multiplication using three kinds of arithmetics, in which CRT-MOD, ICRT-G, NTT-MM and INTT-MM are adopted and element-wise multiplication (EWM) is also included. The results are shown in Fig. 4.

In general, the INT64 implementation outperforms FP64/INT32 implementation by 8%/11% on A100. However, it works poorly on P100. On the one hand, P100 GPU has low reading and writing efficiency for INT64 numbers.

TABLE II: Performance comparison of homomorphic multiplication implementations

Implementation	Device	TFLOPS	Execution time (ms)					Total time ×
			CRT	NTT	INTT	ICRT	Total	(TFLOPS / 16.31)
HEAAN [31]*	24-threads CPU	/	35.9	24.4	24.8	59.4	154.5	-
AVX-512 Imp. [15]	24-threads CPU	/	12.3	7.0	7.4	37.2	74.8	-
GPU Imp. [15]	Titan RTX GPU	16.31	4.1	3.7	4.7	19.4	38.1	38.1
Our FP64 Imp.	Tesla P100 GPU	9.53	5.5	5.0	7.4	40.9	65.7	37.9
Our INT64 Imp.	A100 GPU	19.49	0.9	1.1	1.2	4.6	9.9	11.8

*: Results are from [15].

On the other hand, INT64 multiplications require multiple native INT32 multiplications, while INT32 multiplication is performed by multiple native instructions on P100. FP64 arithmetic, which also uses INT64 arrays to store moduli, performs well on P100. Compared with INT64/INT32 implementation, FP64 implementation has a 26%/27% performance advantage.

C. Related Work Comparison

In this subsection, we compare our FP64 implementation on Tesla P100 and INT64 implementation A100, adopting CRT-MOD, ICRT-G, NTT-MM and INTT-MM, with state-of-the-art CPU implementations and GPU implementations respectively.

TABLE III: Performance comparison of our work and CPU implementations using a polynomial degree of 2^{15}

Implementation	$(\log_2 q_i, r)$	Execution time (μs) / Speedup#		
		NTT	INTT	EWM
Native C++*	$(60, 1)$	368	374	44.9
Intel AVX512-DQ*	$(60, 1)$	131 (2.8×)	140 (2.7×)	25.7 (1.7×)
Our FP64 Imp. on P100	$(49, 18)$	117 (57×)	120 (56×)	36.4 (22×)
Our INT64 Imp. on A100	$(60, 15)$	33 (167×)	35 (160×)	9.7 (69×)

*: Intel HEXL kernel [40] running on Intel i9-10940X CPU.
#: The speedup is compared with native C++ implementation and multiplied by the ratio of modulus number (r).

1) vs. CPU implementations: In Table III, we compare our implementations, processing multiple (18/15) sets of polynomial arithmetic to make use of massive computing units, with the CPU implementations. Compare with native C++ implementation, our implementation on A100 have over 160 times performance advantage for NTT and 69 times improvement for element-wise ModMul. Compared with the highly optimized Intel HEXL kernel [40] with Intel AVX512-DQ instructions, our implementation on A100 are 71-72× faster for NTT and 47× faster for element-wise ModMul. 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.

TABLE IV: Performance comparison of CRT and NTT implementations on Tesla P100

Imp.	$N = 2^{14}$, $\log_2 q \approx 744$			$N = 2^{15}$, $\log_2 q \approx 881$		
	[41]	Our FP64 Imp.	Speedup	cuHE*	Our FP64 Imp.	Speedup
CRT(μs)	218	66	3.3×	705	81	8.7×
NTT(μs)	87	52	1.7×	623	117	5.3×
INTT(μs)	93	56	1.7×	573	120	4.8×
ICRT(μs)	323	100	3.2×	586	277	2.1×

*: cuHE library [42] running on Tesla P100.

2) vs. GPU implementations: In Table IV, we compare our CRT and NTT implementations with other GPU implementations running on Tesla P100 GPU card. The introduction of FP64 arithmetic and a series of optimizations of polynomial arithmetic enabled our implementations to achieve significant performance advantages. For better comparison with [41], we choose a set of parameters used in it. Our CRT/ICRT implementations achieve over 3.2-3.3× speedup and our NTT/INTT implementations achieve 1.7× speedup over their work. Compared with the cuHE library [42], our advantage is more obvious. Our CRT/ICRT implementations outperform by 2.1× to 8.7×, and our NTT/INTT implementations outperform by 4.8× to 5.3×.

3) Homomorphic multiplication comparison: We integrated our polynomial arithmetic into HEAAN v2.1 library [31] and compare the homomorphic multiplication of CKKS scheme with the implementations using the same library. The same experimental setup as in [31] ($\log_2 N = 16$, $\log_2 q = 1200$) are adopted in our implementation. We assume that multiple homomorphic operations are needed for the ciphertext, and that all polynomial arithmetics are carried out by GPU kernels. In this case, the memory transfer delay is negligible when considering a single homomorphic multiplication. For some operations, the four polynomials involved can be processed in parallel, so we use four streams to maximize the utilization of GPU. Table II compares the homomorphic multiplication of our work and previous work [15]. Compared with the original HEAAN v2.1 library and the AVX-512 accelerated implementation running on a 24-thread CPU, our implementation on A100 is 15.7× to 7.6× faster, respectively. When comparing results on different GPU platforms, the performance difference of hardware is taken into account by multiplying the ratio of TFLOPS values. Although P100 is two generations behind Titan RTX, our implementation on it still achieves the same level execution time of the GPU implementation in [15]. Powered by highly optimized polynomial arithmetic and underlying arithmetic, our implementation on A100 achieves 3.2× speedup over the GPU implementation in [15].

VI. 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 are implemented based on two arithmetic units with three types of data precision. A case study with CKKS as an example demonstrates that all the three series of our implementations outperform the state-of-the-art GPU-based implementation by 3.2×. 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, and apply the GPU-based FHE schemes to real-world workloads, e.g., high-performance privacy-preserving deep learning workloads.

REFERENCES

[1] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

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

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

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

[7] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A graph compiler for deep learning on homomorphically encrypted data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 3–13.

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

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

[10] Microsoft, "Microsoft SEAL Release 3.7.2," https://github.com/Microsoft/SEAL/releases/tag/v3.7.2, 2022.

[11] S. H. et al., "HElib v2.2.1," https://github.com/homenc/HElib, 2021.

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

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

[14] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

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

[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 Cryptol. ePrint Arch.*, vol. 2021, p. 508, 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," *Journal of Parallel and Distributed Computing*, vol. 119, pp. 179–190, 2018.

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

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

[20] O. Ozerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 124, 2021.

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

[22] J. Dong, F. Zheng, N. Emmart, J. Lin, and C. Weems, "sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digital signature computations," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 599–609.

[23] L. Gao, F. Zheng, R. Wei, J. Dong, N. Emmart, Y. Ma, J. Lin, and C. Weems, "DPF-ECC: A framework for efficient ECC with double precision floating-point computing power," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3988–4002, 2021.

[24] R. Wei, F. Zheng, L. Gao, J. Dong, G. Fan, L. Wan, J. Lin, and Y. Wang, "Heterogeneous-PAKE: Bridging the gap between PAKE protocols and their real-world deployment," in *Annual Computer Security Applications Conference*, 2021, pp. 76–90.

[25] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[26] F. Winkler, *Polynomial algorithms in computer algebra*. Springer Science & Business Media, 1996.

[27] R. Crandall and B. Fagin, "Discrete weighted transforms and large-integer arithmetic," *Mathematics of computation*, vol. 62, no. 205, pp. 305–324, 1994.

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

[29] I. S. Committee *et al.*, "754-2008 IEEE standard for floating-point arithmetic," *IEEE Computer Society Std*, vol. 2008, 2008.

[30] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Annual Cryptology Conference*. Springer, 2012, pp. 850–867.

[31] "Release heaan with faster multiplication · snucrypto/heaan," https://github.com/snucrypto/HEAAN/releases/tag/2.1, (Accessed on 01/22/2023).

[32] L. Wan, F. Zheng, G. Fan, R. Wei, L. Gao, Y. Wang, J. Lin, and J. Dong, "A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator," in *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Springer, 2022, pp. 514–534.

[33] A. Al Badawi, B. Veeravalli, and K. M. M. Aung, "Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution," in *Future of Information and Communication Conference* Springer, 2018, pp. 666–682.

[34] 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 Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 941–956, 2019.

[35] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques* Springer, 1986, pp. 311–323.

[36] V. Shoup, "NTL: A library for doing number theory," https://libntl.org/, (Accessed on 01/22/2023).

[37] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

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

[39] H. L. Garner, "The residue number system," in *Papers presented at the the March 3-5, 1959, western joint computer conference*, 1959, pp. 146–153.

[40] "intel/hexl: Intel homomorphic encryption acceleration library accelerates modular arithmetic operations used in homomorphic encryption," https://github.com/intel/hexl, (Accessed on 01/29/2023).

[41] 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 Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, pp. 70–95, 2018.

[42] "vernamlab/cuhe: CUDA homomorphic encryption library," https://github.com/vernamlab/cuHE, (Accessed on 01/22/2023).