HE-Booster: An Efficient Polynomial Arithmetic Acceleration on GPUs for Fully Homomorphic Encryption

Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng

Abstract—Fully Homomorphic Encryption (FHE) enables secure offloading of computations to untrusted cloud servers as it allows computing on encrypted data. However, existing well-known FHE schemes suffer from heavy performance overheads. Thus numerous accelerations based on FPGAs, ASICs, and GPUs have been proposed. Compared to FPGAs and ASICs, GPUs have obvious advantages in productivity and development costs. And also, GPUs have already been widely deployed in commercial cloud or supercomputing centers. Therefore, we present HE-Booster, an efficient GPU-based FHE acceleration design. For single-GPU acceleration, a thorough systematic design is exploited to map five common phases in typical FHE schemes to the GPU parallel architecture. In particular, inspired by the regular architecture of NTT/INTT, a novel inter-thread local synchronization is proposed to exploit thread-level parallelism. For multi-GPU acceleration, we propose a scalable parallelization design that exploits *data-level parallelism* through fine-grained data partition under different representations. Finally, experiments on 1 NVIDIA GPU demonstrate that our work outperforms 251.7×, 78.5× and 164.9× than three mainstream CPU-based libraries HElib, SEAL, and PALISADE, and up to 170.5× speedup is obtained compared to the GPU-accelerated library cuHE. What's more, performing 8 homomorphic multiplications on 8 GPUs can deliver up to a 7.66× performance boost compared to a single-GPU implementation.

Index Terms—Fully homomorphic encryption, GPU acceleration, number-theoretic transform

1 Introduction

As cloud computing becomes more widespread, it also opens up potential security and privacy risks due to the need to access raw data [1], [2], [3]. Fully homomorphic encryption (FHE) is considered a promising technology to address these challenges as it allows computing on encrypted data (ciphertext). Specifically, it enables the client's computation to be securely offloaded to an untrusted server, which directly processes the ciphertext without decryption and returns the encrypted result to the client. The decryption of the resulting ciphertext matches the result of performing the same computation on the unencrypted data (plaintext). In this way, clients can leverage the computing power of the cloud to gain meaningful insights from the data while effectively protecting the confidentiality of the data. Furthermore, compared to partially homomorphic encryption (PHE) that only supports addition or multiplication (e.g., Paillier[4], RSA[5] and ElGamal[6]), FHE allows arbitrary computation on ciphertext, which motivates a great deal of research work[7], [8], [9], [10], [11].

However, FHE is not yet widely deployed due to its significant performance overhead. Ciphertext computation can introduce more than 4 orders of magnitude slowdown compared to plaintext computation. In typical FHE schemes like BGV[12], BFV[13], [14] and CKKS[15], the major performance bottleneck comes from massive polynomial arithmetic. Specifically, encrypted data generally consist of a pair of polynomials, whose coefficients reach hundreds or thousands of bits, demanding expensive multi-word arithmetic. Moreover, a large polynomial length exacerbates the computational complexity, especially for polynomial multiplication. Therefore, the essence of FHE acceleration lies in accelerating polynomial arithmetic.

The prevalent acceleration strategy is offloading time-consuming polynomial arithmetic to hardware accelerators (e.g., FPGAs, ASICs, and GPUs). Several studies propose kinds of acceleration mechanisms based on FPGAs. They use fixed-size parameter sets that limit the FHE configurations in diverse application scenarios [16], [17], [18]. Moreover, such FPGA-based solutions face the challenge of productivity. Alternatively, F1 [19], CraterLake[20], BTS [21], ARK[22] and BASALISC[23] respectively designs ASIC accelerator architectures that target different optimizations, but they have only been implemented on simulators so far.

- *Zhiwei Wang and Zhihao Li are with the State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing 100093, China, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing 101408, China. E-mail: {wangzhiwei, lizhihao}@iie.ac.cn.*
- *Peinan Li, Rui Hou, Jiangfeng Cao, and Dan Meng are with the State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing 100093, China. E-mail: {lipeinan, hourui, caojiangfeng, mengdan}@iie.ac.cn.*
- *XiaoFeng Wang is with Indiana University at Bloomington, Bloomington, IN 47405 USA. E-mail: xw7@indiana.edu.*

Manuscript received 5 July 2022; revised 1 November 2022; accepted 7 December 2022. Date of publication 9 January 2023; date of current version 10 February 2023.
This work was supported in part by the National Natural Science Fund of China for Distinguished Young Scholars under Grant 62125208 and in part by the Young Scientists Fund of the National Natural Science Foundation of China under Grant 62202467.
(Corresponding author: Rui Hou.)
Recommended for acceptance by H. Huang.
Digital Object Identifier no. 10.1109/TPDS.2022.3228628

Furthermore, implementing such kind of ASIC chip requires huge development costs. By contrast, GPU is a better candidate since there exists a series of commercial and off-the-shelf products. Therefore, it is worth investigating an efficient GPU-based FHE acceleration design.

Typically, polynomial arithmetic acceleration in FHE consists of five phases, involving the Chinese Remainder Theorem (CRT) and Number-Theoretic Transform (NTT) for the forward transformation, Operation phase, and INTT (inverse NTT) and ICRT (inverse CRT) for the backward transformation. As a well-known state-of-the-art GPU-based polynomial arithmetic library, cuHE [24] provides a typical implementation, including efficient algebraic tools and memory management. However, there exist two main limitations: On the one hand, it only supports single-word CRT moduli (less than 32-bit) for polynomial arithmetic. Fewer bit representations fail to exploit potential parallel sub-spaces, limit the configurability of FHE programs (e.g., multiplicative depth), and reduce performance (e.g., more intensive computations). Similar problems are observed in other solutions[25], [26], [27]. On the other hand, cuHE adopts the *cyclic convolution*-based four-step Fast Fourier Transform (FFT) algorithm [28] to implement NTT. However, it requires polynomial length doubling and additional polynomial reduction operation, causing heavy computation overheads. Another study[29] reduces the overhead through a GPU global synchronization-based NTT with *negative wrapped convolution*. However, it does not fully exploit thread-level parallelism, causing long synchronization delays and hurting performance.

In this paper, we present HE-Booster, an efficient GPU-based FHE acceleration design. The primary contributions are listed as follows:

- *For single-GPU acceleration, we propose a thorough systematic design to map five common phases in typical FHE schemes to the GPU architecture.***(i)** Performance-critical NTT/INTT operations involve multiple stages, and these stages proceed sequentially due to data dependencies. In conventional design, a thread cannot be executed until all threads in the previous stage are completed, causing a long synchronization delay. Specifically, the *coarse-grained global synchronization* can be translated into *local synchronization* of massive fine-grained thread pairs, that is, a thread of the current stage depends only on another thread of the previous stage in addition to itself. Moreover, this regularity provides an opportunity to identify stable and ordered dependencies between threads. Therefore, HE-Booster employs a *synchronization trigger* to capture the dependency precisely and adopts *inter-thread local synchronization* to allow threads to be executed as early as possible. Experiments demonstrate that local synchronization introduces up to 1.9× performance boost for a single NTT execution. **(ii)** For CRT and ICRT phases, we propose *transpose* to improve the memory access efficiency of the coefficients matrix for the first time. **(iii)** For Operation phase, we utilize *kernel fusion* to improve GPU resource utilization and reduce overall launch overhead effectively.
- *For multi-GPU acceleration, we propose a scalable parallelization design that exploits fine-grained data-level parallelism.*We observe that homomorphic operations majorly involve numerous calculations of polynomials in different representations (e.g., CRT and NTT representations.). These polynomials are calculated from original coefficient matrices. And the size of each matrix is large and can thus be partitioned for parallel execution. Therefore, in the case of the acceleration on multiple GPUs, the opportunities come from the tremendous *data-level parallelism* provided by those large coefficient matrices, which can be exploited to map the same task on multiple GPUs to process different data partitions. Since these coefficient matrices are usually processed in different patterns or granularity, three typical data partitions strategies are adopted accordingly at different stages of homomorphic multiplication.
- *We implement HE-Booster based on CUDA and evaluate overall performance on realistic NVIDIA GPU server*. For single-GPU implementation, HE-Booster can produce 251.7×, 78.5× and 164.9× speedup compared to three CPU-based libraries HElib, SEAL, and PALISADE. Meanwhile, up to 170.5× performance improvement is achieved compared to the GPU-based library cuHE. For multi-GPU acceleration, implementing 8 homomorphic multiplications on 8 GPUs can deliver up to 7.66× performance improvement compared to a single GPU. Furthermore, we implement the encrypted inference with a plaintext CNN model and encrypted MNIST images based on HE-Booster. Compared to the state-of-the-art HCNN [10] with 1 GPU, running such encrypted inference with HE-booster on 1 GPU achieves 1.23× speedup. And this speedup increases to 4.3× on 8 GPUs.

2 Background

In typical FHE schemes, BGV[12] and BFV[13], [14] perform exact arithmetic on integer, and CKKS[15] supports approximate arithmetic on complex or real numbers. They are all built on the ring-learning-with-errors (RLWE) assumptions to guarantee security[30]. Although these schemes differ in their encryption methods, they all use the same data structure (e.g., *polynomial*) to represent ciphertext. It means the parallel algorithms used to speed up computation are similar. Moreover, their fundamental homomorphic operations are similar and share common ciphertext maintenance techniques (e.g., *key switching*). Therefore, these commonalities lend us an opportunity to present a general FHE acceleration design. For concreteness, we use the BGV scheme as the example to introduce FHE in Section 2.1. Besides, we describe two popular parallel acceleration algorithms in Section 2.2.

2.1 BGV Scheme

In BGV scheme, the plaintext modulus t and ciphertext modulus Q determine plaintext space $R_t = \mathbb{Z}_t/(X^N + 1)$ and ciphertext spaces $R_Q = \mathbb{Z}_Q/(X^N + 1)$ respectively, where $Q \gg t$ and N is a power of 2. For example, BGV encodes each plaintext vector into a polynomial $\mathfrak{m} \in R_t$, like

$$\mathfrak{m} = m_0 + m_1 x + \cdots + m_{N-1} x^{N-1} \in R_t.$$

Each plaintext is encrypted into a ciphertext consisting of a pair of polynomials, and each polynomial contains N coefficients modulo Q. Note that all polynomial operations are modulo Q and $X^N + 1$.

Encryption and Decryption: The *secret key* is uniformly sampled in $\{-1, 0, 1\}$, denoted as $\mathfrak{s} \leftarrow \mathcal{X}_{\text{key}}$. To encrypt a plaintext $\mathfrak{m}$, it samples a uniformly random $\mathfrak{a} \leftarrow R_Q$, an error $\mathfrak{e} \leftarrow \mathcal{X}_{\text{err}}$ and computes the ciphertext $\mathbf{ct}$ as

$$\mathbf{ct} = (\mathfrak{b}, \mathfrak{a}) = (-\mathfrak{a}\mathfrak{s} + t\mathfrak{e} + \mathfrak{m}, \mathfrak{a}).$$

To decrypt the ciphertext $\mathbf{ct}$, it computes $\mathfrak{b} + \mathfrak{a} \cdot \mathfrak{s}$ and modulo t to recover plaintext $\mathfrak{m}$.

Homomorphic addition (HEADD) of two ciphertexts $\mathbf{ct_0} = (\mathfrak{b}_0, \mathfrak{a}_0)$ and $\mathbf{ct_1} = (\mathfrak{b}_1, \mathfrak{a}_1)$ can be achieved by the polynomial-wise addition: $\mathbf{ct}_{\text{add}} = (\mathfrak{a}_0 + \mathfrak{a}_1, \mathfrak{b}_0 + \mathfrak{b}_1)$.

Homomorphic multiplication (HEMUL) reqiures two steps. First, the tensor product of two ciphertext is computed as follows:

$$\mathbf{ct}_\times = (\mathfrak{c}_0, \mathfrak{c}_1, \mathfrak{c}_2) = (\mathfrak{b}_0\mathfrak{b}_1, \mathfrak{a}_0\mathfrak{b}_1 + \mathfrak{a}_1\mathfrak{b}_0, \mathfrak{a}_0\mathfrak{a}_1).$$

The ciphertext $\mathbf{ct}_\times$ can be regarded as a intermediate result under a different secret key. Second, a critical ciphertext maintenance technique, called *key switching*, is employed to recover the original 2-dimension result. Specifically, two key switching implementations can be adopted.

In the Brakerski-Vaikuntanathan (BV) variants[13], *key-switching matrices* are computed: $\mathsf{ksm} = (\mathsf{ksm}[k][0], \mathsf{ksm}[k][1]) = (-\mathfrak{a}_k\mathfrak{s} + t\mathfrak{e}_k + \omega^k\mathfrak{s}^2, \mathfrak{a}_k)$, where radix $\omega \ll Q$, $l_{\omega,Q} = \lfloor \log_\omega Q \rceil + 1$ and $0 \le k < l_{\omega,Q}$. It first performs digit decomposition on $\mathfrak{c}_2$ to produce $\mathfrak{c}_2^{(k)}$: $\mathfrak{c}_2 = \sum_{k=0}^{l_{\omega,Q}-1} \mathfrak{c}_2^{(k)} \omega^k$. Then, the multiplicative accumulation (MAC) operation is computed to produce

$$(\mathfrak{l}_0, \mathfrak{l}_1) = \left(\sum_{k=0}^{l_{\omega,Q}-1} \mathsf{ksm}[k][0] \cdot \mathfrak{c}_2^{(k)}, \sum_{k=0}^{l_{\omega,Q}-1} \mathsf{ksm}[k][1] \cdot \mathfrak{c}_2^{(k)} \right)$$

and the final ciphertext result can be computed: $\mathbf{ct}_{\text{mul}} = (\mathfrak{c}_0 + \mathfrak{l}_0, \mathfrak{c}_1 + \mathfrak{l}_1)$.

In the Gentry-Halevi-Smart (GHS) variant [31], it needs to temporarily extend the size of Q with another modulus P and generates the *key-switching keys* $\mathsf{ksk} = (\mathsf{ksk}_1, \mathsf{ksk}_2) = (\mathfrak{b}', \mathfrak{a}') \in R_{PQ}^2$ where $\mathfrak{a}' \leftarrow R_{PQ}$ and $\mathfrak{b}' = -\mathfrak{a}'\mathfrak{s} + t\mathfrak{e}' + P\mathfrak{s}^2 \in R_{PQ}$. Then it performs

$$(\mathfrak{l}_0, \mathfrak{l}_1) = (\lfloor P^{-1} \cdot \mathfrak{c}_2 \cdot \mathsf{ksk}_1 \rceil, \lfloor P^{-1} \cdot \mathfrak{c}_2 \cdot \mathsf{ksk}_2 \rceil)$$

and the final ciphertext result can be computed: $\mathbf{ct}_{\text{mul}} = (\mathfrak{c}_0 + \mathfrak{l}_0, \mathfrak{c}_1 + \mathfrak{l}_1)$. Besides, a hybrid version combining both methods is proposed to balance the tradeoff between the complexity and the modulus increase [32], [33], [34].

Homomorphic Rotation (HEROT) provides a complete set of operations over packed plaintext vectors in the encrypted form [35]. Generally, in the HEROT implementation, an *automorphism* should be computed first on the packed ciphertext, which can be expressed as $\mathbf{ct}_\sigma = (\sigma_k(\mathfrak{b}), \sigma_k(\mathfrak{a}))$. Specifically,

$$\sigma_k(\mathfrak{a}) : a_i \rightarrow (-1)^s a_{ik \bmod N} \text{ for } i = 0, 1, ..., N-1$$

where k is a positive odd less than N and $s = 0$ if $ik \bmod 2N < N$, and $s = 1$ otherwise. Like HEMUL, HEROT requires a key switching process to compute the final results: $\mathbf{ct_{rot}} = (\mathfrak{l}_0 + \sigma_k(\mathfrak{b}), \mathfrak{l}_1)$, where $(\mathfrak{l}_0, \mathfrak{l}_1) = \text{KeySwitching}(\sigma_k(\mathfrak{a}))$.

Besides, *modulus switching* technique can avoid the noise explosion during the computation. It usually switches the ciphertext from the current space R_Q to a smaller space $R_{Q'}$, achieving a proportional noise reduction, which can be expressed as $\mathbf{ct}' = \lfloor \frac{Q'}{Q} \mathbf{ct} \rceil$.

Based on the above overview, we can see that FHE involves massive polynomial arithmetic, which becomes the primary performance bottleneck.

2.2 Parallel Algorithms

Chinese Remainder Theorem (CRT): Translate expensive multi-word arithmetic into parallel lightweight polynomial vector operations. FHE usually requires a wide ciphertext space R_Q with hundreds or thousands of bits, causing expensive multi-word arithmetic. CRT addresses this problem by reducing multi-word polynomial coefficients to multiple independent small-integer polynomial vectors. For instance, the ciphertext modulus Q is usually defined as the product of distinct primes $Q = \prod_{i=1}^{L} q_i$, where q_i fits in a single or double word. Then, a polynomial in R_Q can be represented as L polynomials in sub-space $R_{q_1}, \cdots, R_{q_L}$. As a result, CRT translates a complex multi-word polynomial arithmetic into L parallel lightweight computations. Besides, there is an obvious observation that the bit-width of the q_i will affect the number L of parallel sub-spaces.

Number-Theoretic Transform (NTT): Convert polynomial vectors into point-valued representations, reducing multiplication complexity from $O(N^2)$ to $O(N\log N)$. Polynomial multiplication is common in cryptographic algorithms. Using conventional *convolution coefficient* method induces $O(N^2)$ time complexity. Thus, NTT [36], a special generalization of the FFT [28], can be applied to reduce computation complexity to $O(N\log N)$. Specifically, NTT converts a length-N polynomial vector into its point-valued representation. Then, polynomial multiplication can be achieved by element-wise multiplication in the NTT domain. This process can be expressed as $\mathsf{NTT}(\mathfrak{a}\mathfrak{b}) = \mathsf{NTT}(\mathfrak{a}) \odot \mathsf{NTT}(\mathfrak{b})$, where $\odot$ denotes element-wise multiplication (i.e., hadamard product). There are two methods to implement NTT: *cyclic convolution* and *negative wrapped convolution*. Compared to the former, the latter has better performance since it can eliminate the polynomial length doubling. It is worth noting that polynomials represented in the NTT domain can avoid unnecessary transformations across operations. It should be noted that the NTT is a linear transformation, so addition and automorphism operations can also be computed in the NTT domain.

3 HE-Booster on a Single GPU

3.1 Working Flow of HE-Booster

To accelerate FHE, it is natural to introduce parallel algorithms into GPU many-core architectures. Moreover, the acceleration process of ciphertext computation is essentially an ordered *dataflow graph*, and its critical path can be divided into five phases: CRT and NTT for the forward transformation, FHE operations, INTT and ICRT for the backward transformation [24], [25], [26], [37]. Therefore,

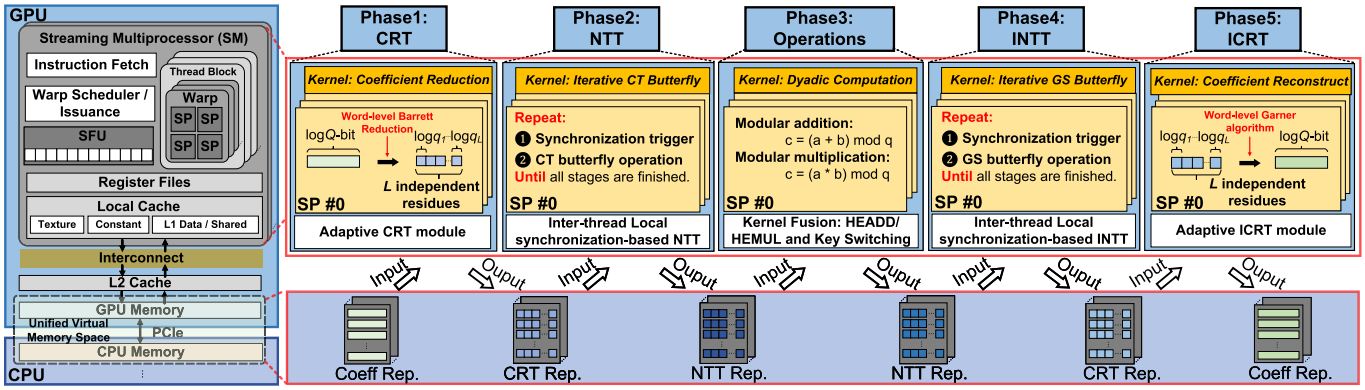

Fig. 1. Working flow of HE-Booster on a single GPU.

this paper focuses on these five phases to build an efficient FHE acceleration design.

Typically, in a GPU hardware architecture and programming model, the GPU kernel execution utilizes massive threads to process data in parallel. These threads are organized into multiple thread blocks (TBs), and TBs are assigned to Streaming Multiprocessors (SMs). Different TBs can share the same SM, but each TB can only be executed within a single SM. Further, threads in a TB are grouped into multiple warps, which are mapped to successive Streaming Processors (SPs). Then, these warps run in an arbitrary order managed by the warp scheduler. Data required in programs are moved from CPU memory to GPU global memory (GMEM), and then to L2 cache, L1 cache, and registers successively. Frequent access data may be loaded into shared memory (SMEM) for better performance.

Following the above design principles, we can efficiently map five phases to the GPU architecture. Fig. 1 shows our design overview. In the working flow of HE-Booster, five phases have their kernels and are executed sequentially. First, CRT kernels take coefficient representations (Coeff Rep.) and convert them to CRT representations (CRT Rep.). Then, CRT Rep. is consumed by NTT kernels and converted into NTT representations (NTT Rep.). After FHE computation in the *Operation* phase, the resulting NTT Rep. is recovered to Coeff Rep. through INTT and ICRT phases. We will describe each phase in detail below.

Phase 1: CRT decomposes ciphertext space into multiple independent sub-spaces, providing an opportunity to utilize the GPU parallel architecture. Specifically, each thread in the CRT kernel performs *coefficient reduction* operation to transform multi-word coefficients in ciphertext space into multiple independent single- or double-word residues. HE-Booster focuses on optimizing memory access efficiency in this phase and introduces *transpose* optimization to improve the memory access efficiency of the coefficient matrix for the first time. Specific design is shown in Section 3.2.

Phase 2: NTT is processed in multiple stages, in which each thread performs *Cooley-Tukey* (CT) butterfly [38] of different coefficient pairs. There exist strict data dependency between each stage. Conventional design [29], [39] employs coarse-grained global synchronization to guarantee the correctness. In this case, a thread cannot be executed until all threads in the previous stage are completed, causing long synchronization delays. Fortunately, we find that the NTT implementation exhibits regular locality. Specifically, stage-to-stage global synchronization can be translated into local dependencies of massive fine-grained thread pairs, that is, a thread of the current stage depends only on another thread of the previous stage in addition to itself. Moreover, this regularity enables us to identify stable and ordered dependencies between threads. Inspired by this observation, we propose an efficient *inter-thread local synchronization* and employ *synchronization trigger* to capture the dependence precisely, allowing threads to be executed as early as possible. Details are shown in Section 3.3.

Phase 3: Operation performs dyadic computation on vectors in the NTT domain. Specifically, each thread computes element-wise modular addition and multiplication according to specific computing logic. The computationally intensive *key switching* process entails performing dyadic kernels many times, resulting in dramatic launch overheads. As a widely used technique to improve computing performance, *kernel fusion* can fuse small-size kernels that are frequently launched into large-size one that only need to be launched once. It can significantly reduce launch overhead and improve memory access efficiency. Section 3.4 demonstrates the details.

Phase 4: INTT uses the same synchronization mechanism as NTT. It is slightly different from NTT in that each thread computes *Gentleman-Sander* (GS) butterfly operations[40].

Phase 5: ICRT reconstructs multiple residue polynomials (independent sub-spaces) into a single polynomial (ciphertext space) with multi-word coefficients. In the ICRT kernel, each thread will performs *coefficient reconstruction* operation to recover a ciphertext polynomial coefficient.

3.2 CRT and ICRT Phases Acceleration With Memory Access Optimizations

HE-Booster focuses on optimizing memory access efficiency, and also optimizes the transformation efficiency and calculation efficiency with classic approaches in the CRT and ICRT phases. Specifically, In *CRT phase*, each coefficient of a polynomial in ciphertext space ($Q = \prod_{i=1}^{L} q_i$) is performed with a modulo operation. According to the representation of x,

$$x = x_1(2^{64})^0 + x_2(2^{64})^1 + \cdots + x_d(2^{64})$$

a residue vector $\{x \bmod q_1, x \bmod q_2, \cdots, x \bmod q_L\}$ is generated. *ICRT phase* reconstructs multiple parallel sub-spaces into original ciphertext space.

Introduce transpose to improve memory access efficiency of coefficients matrix for the first time. During the CRT phase, all source

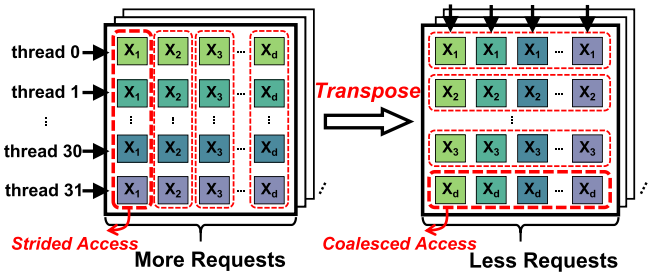

Fig. 2. Coefficient matrix transpose in CRT.

data locates in coefficients matrix, and each thread frequently loads corresponding x_k $(1 \leq k \leq d)$ from coefficients matrix and then x_k is mod by CRT moduli q $(\{q_1, q_2, \cdots, q_L\})$. In a typical NVIDIA GPU design, 32 threads are grouped to a warp and all their memory accesses share the same load port. Notably, 32 threads in a warp require to load 32 x_k for parallel execution. Unfortunately, the 32 x_k in the coefficient matrix is not contiguous and they have to be loaded in the strided style, as shown in Fig. 2. For instance, when d equals 8 and the cache line size is 64B, loading 32 double-word x_k in a warp requires 32 memory requests, inducing severe pipeline stalls. To address this issue, when these x_k are loaded from GMEM to low-latency SMEM [41], [42], we *transpose* the coefficients matrix. After the transpose operation, accesses to x_k is changed from column to row. For subsequent parallel execution, the x_k in different threads of the same warp belong to the same row, and their access efficiency can be significantly improved since their addresses are contiguous [43].

Employ classical algorithms to improve transformation efficiency. As for transformation from ciphertext space to sub-spaces in the CRT phase, we employ the *Barrett Reduction* [44] to replace costly modulo with lightweight multiplication, shifting, and subtraction. During the transformation from sub-spaces to the ciphertext in the ICRT phase, *Garner's algorithm* [45] is utilized to decrease the number of modulo operations through mixed-radix representation. This implementation is similar to existing acceleration mechanisms [25], [27].

Implement double-word arithmetic to improve calculation efficiency. CRT moduli can be represented as 32-bit or 64-bit integers in typical CPU-based FHE designs [32], [46], [47]. However, in state-of-the-art GPU-based acceleration schemes [24], [25], [27], [48], [49], only 32-bit (single-word) representation is implemented. In our design, we utilize carry operation instructions (e.g., addc, subc, and madc) in CUDA PTX assembly language [50] to implement the fundamental arithmetic. With these instructions, multi-word arithmetic can be processed as efficiently as computation without emulation [48]. Specifically, through implementing the performance-critical double-word shifting operation in *Barrett Reduction*, more instruction-level parallelism and high efficiency can be achieved. Furthermore, both mixed-radix modulo operations and lightweight accumulation methods are supported in the *Garner's algorithm*.

3.3 NTT and INTT Phases Acceleration With Inter-Thread Local Synchronization

An NTT operation is processed in multiple stages. Specifically, an N-point NTT operation has $\log N$ stages, and these

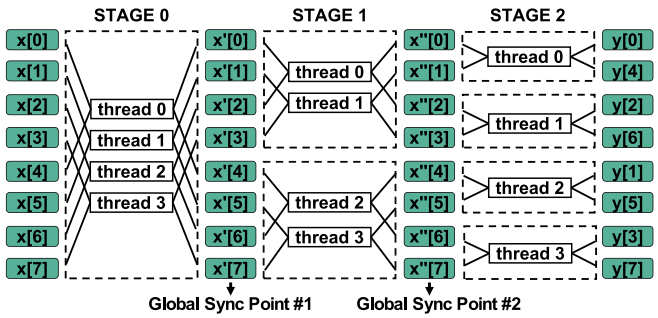

Fig. 3. 8-point NTT with inter-block global synchronization.

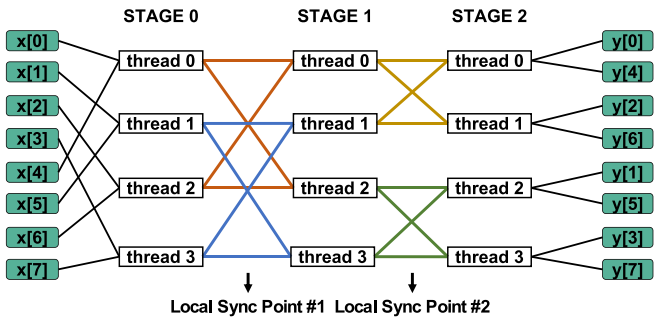

Fig. 4. True inter-thread dependency in 8-point NTT. Threads connected with lines in same color form a thread pair.

stages proceed successively due to the data dependencies. Then, stage-to-stage synchronization is necessary.

In the conventional implementations, each thread can be executed only when all threads in the previous stage are completed, noted as ***inter-block global synchronization*** . Fig. 3 shows an example of an 8-point NTT with this method, which executes with four GPU threads, and each thread processes two elements at each stage. There are several representative methods in the literature: On the CPU side, the synchronization can be achieved through launching kernel $\log N$ times, which induces substantial launch overheads [29]. Alternatively, on the GPU side, the synchronization can be achieved through a global synchronization function inside the kernel [39]. However, coarse-grained global synchronization causes a long waiting delay and hurts performance.

Fortunately, we find that NTT and INTT implementations have regular locality, and the coarse-grained global synchronization can be translated into local synchronization of massive fine-grained thread pairs. In this way, a thread in the current stage only depends on another thread in the previous stage in addition to itself. Moreover, this regularity makes the dependencies between threads quite stable and orderly. As shown in Fig. 4, the true inter-thread dependencies are extracted from Fig. 3. For instance, thread 0/2 in stage 1 only depends on thread 0 and thread 2 in stage 0. Thread 0/1 in stage 2 only depends on thread 0 and thread 1 in stage 1. A similar dependence relationship can be observed for other threads. In conclusion, a thread can be executed when corresponding interdependent threads (noted as *thread pair*) are finished. As long as the interdependent *thread pair* can be identified precisely for each thread, a thread can be executed as early as possible. Fig. 5 illustrates the inter-block global synchronization and inter-thread local synchronization in principle. Local synchronization decreases the waiting time for global synchronization, leading to a remarkable performance gain.

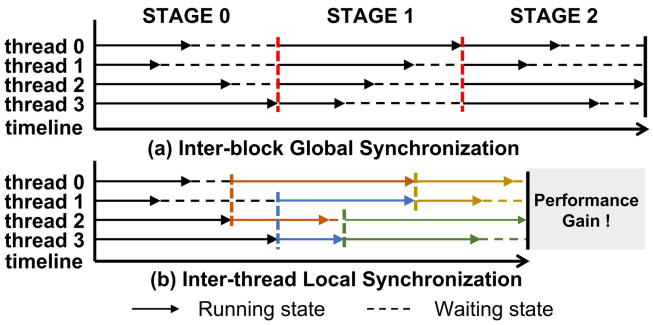

Fig. 5. Execution time comparison between inter-block global synchronization and inter-thread local synchronization.

To support the local synchronization, it is necessary to precisely identify thread pairs that change dynamically in different stages. We allocate a *synchronization trigger vector* along with the intermediate data, such as the array x′ and x″ in Fig. 3. Each bit in this vector indicates whether the corresponding data is ready for usage in the subsequent stage. For instance, when thread 0 in stage 0 finishes, trigger bits corresponding to x′[0] and x′[4] are set. Then, thread pairs can be identified according to the data required for a thread in subsequent stages. As for thread 0 in stage 1, it can be executed when x′[0] and x′[2] are ready. Then, as long as threads 0 and 2 in stage 0 finish and trigger bits related to x′[0] and x′[2] are set, thread 0 in stage 1 can be executed. Algorithm 1 presents the inter-thread local synchronization-based NTT implementation. The conventional global method synchronizes all threads after executing the CT butterfly operation. The local method reverses the two procedures in execution order. Specifically, it identifies and synchronizes the thread pairs by checking the *synchronization trigger vector* (i.e., tri_bit) before computation.

Algorithm 1. Inter-tHread Local Synchronization-Based NTT

Input:
A vector $\mathbf{x}$ in a natural order, Ψ_{rev} stores the powers of ψ in a bit-reversal order, where ψ is called the $2N$-th root of unity satisfing the condition $\psi^{2N} = (1 \bmod q)$.
Output:
A vector $\mathbf{x} \leftarrow \text{NTT}(\mathbf{x})$ in a bit-reversal order.

```
thread_id = Global index of thread in GPU;
for (length = 1; length < N; length *= 2) do
  step = N / length / 2;
  psi_step = thread_id / step;
  arr_idx = 2 · psi_step · step + thread_id % step;
  psi_idx = length + psi_step;
  Psi = Ψ_rev[psi_idx];
  // Wait the trigger bits of corresponding thread pairs in prior stage to be set.
  Local_sync(tri_bit[arr_idx], tri_bit[arr_idx+step]);
  // Perform Cooley-Tukey butterfly operation.
  U = x[arr_idx];
  V = x[arr_idx+step] · Psi (mod q);
  x[arr_idx] = U + V (mod q);
  x[arr_idx+step] = U - V (mod q);
end for
return x
```

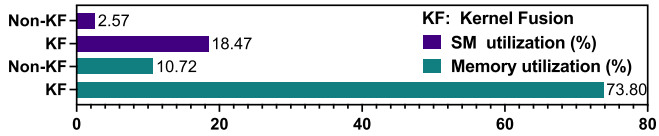

Fig. 6. Hardware utilization evaluation for kernel fusion.

3.4 Operation Phase Acceleration With Kernel Fusion

Homomorphic operations in FHE usually involves massive small-size kernel execution, causing dramatic launch overheads, and each kernel processes a small amount of data, causing huge launch overhead and low GPU resource utilization. Fortunately, enabled with *kernel fusion* technique [51], these kernels can be merged into a single large-size kernel for performance improvement. Fig. 6 shows an example of utilization improvement. For an inner-product of two 32768×50 matrices, launching 100 kernels without kernel fusion (Non-KF) has 2.57% and 10.72% utilization for SM and Memory respectively. After employing kernel fusion, the utilization rates are improved to 18.47% and 73.8% respectively.

Kernel Fusion in Inner Product. Key switching process computes the inner products of decomposed polynomial $\mathfrak{c}_2^{(k)}$ and the key-switching matrix (ksm[k][0], ksm[k][1]) in the BV implementation. Under the NTT representation, the inner product takes the form of $\bar{\mathfrak{l}}_0 = \sum_{k=0}^{l_{\omega,Q}-1} \overline{\text{ksm}}[k][0] \odot \bar{\mathfrak{c}}_2^{(k)}$ and $\bar{\mathfrak{l}}_1 = \sum_{k=0}^{l_{\omega,Q}-1} \overline{\text{ksm}}[k][1] \odot \bar{\mathfrak{c}}_2^{(k)}$. In conventional implementation, 2 ($l_{\omega,Q}$-1) `DyadAdd` and $2l_{\omega,Q}$ `DyadMul` kernels are launched to perform element-wise addition and multiplication. In each `DyadAdd` or `DyadMul` kernel, NL times computations are executed and $4NLl_{\omega,Q}$ times memory accesses are issued. With kernel fusion, all the `DyadAdd` and `DyadMul` kernels can be merged into a single kernel, introducing three benefits: **(i)** Only one kernel launch time is required instead of $4l_{\omega,Q}$-2 kernels. **(ii)** For the repeated access to the same decomposed polynomials $\bar{\mathfrak{c}}_2^{(k)}$, half of the memory accesses can be avoided. **(iii)** Each thread performs $2Ll_{\omega,Q}$ modular multiplications and $2L(l_{\omega,Q}$-1) modular additions instead of only L operations. Then, this method makes full use of GPU resources.

Fig. 7 demonstrates the process to implement the inner product $\bar{\mathfrak{l}}_0$ with and without kernel fusion. In this figure, two $L \times l_{\omega,Q}$ matrices, $\overline{\text{ksm}}[k][0]$ and $\bar{\mathfrak{c}}_2^{(k)}$, are denoted as `KSM0` and `DP` respectively. Fig. 7a shows the traditional method, in which $l_{\omega,Q}$ column-level multiplication and later $l_{\omega,Q}$-1 column-level addition are implemented with independent `DyadMul` and `DyadAdd` kernels. We call this method a simple multiplicative accumulator (SMAC). In contrast, fusing all column-level multiplications and accumulations into only one kernel launch is employed in the Fig. 7b, noted as a fused multiplicative accumulator (FMAC).

Kernel Fusion in Basic Operations. Besides the inner product computation, the kernel fusion technique can also be employed to accelerate basic homomorphic addition and multiplication. Homomorphic multiplication usually requires four `DyadMul` kernel launches and a `DyadAdd` kernel launch. With kernel fusion, a single large-size kernel can be launched to reduce the number of memory accesses from $8N$ to $4N$. In addition, kernel fusion can fuse two `DyadAdd` kernel launches in homomorphic addition into one launch.

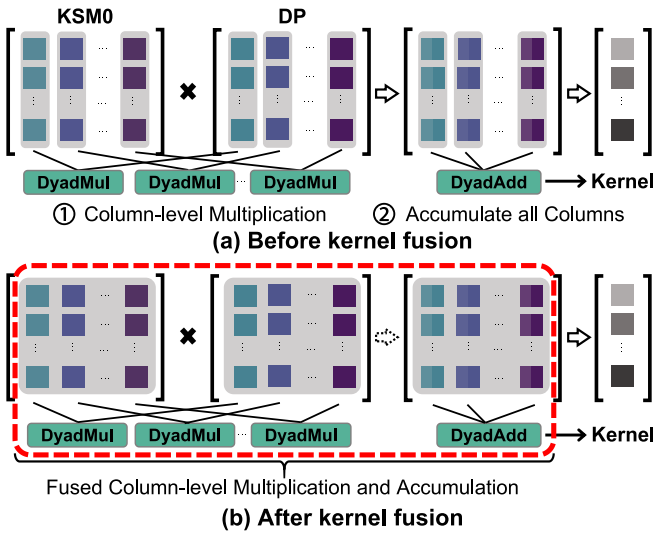

Fig. 7. The inner product implementation of the key switching process with and without kernel fusion.

4 HE-Booster on Multiple GPUs

In supercomputing and cloud scenarios, multiple GPUs are usually integrated with the same high-performance server node. Thus, it is worth investigating the FHE acceleration on multi-GPU systems. A straightforward method is assigning different FHE operations to different GPUs in a operation-level granularity. Essentially, this is a task scheduling mechanism. Instead, this paper focuses on accelerating a single FHE operation on multiple GPUs because such fine-grained parallelization is critical for a scalable design.

Among the homomorphic operations, homomorphic multiplication (HEMUL) and homomorphic rotation (HEROT) suffer from significantly higher computation overhead than homomorphic addition (HEADD). Both HEMUL and HEROT require computationally expensive *key switching* operations to maintain the ciphertext, which dominates the cost of the entire workflow. In contrast, HEADD only involves simple polynomial-wise addition with negligible computation cost. Therefore, it is critical to figure out how to execute *key switching* operations in parallel on multiple GPUs for overall performance gain. Further, due to the commonality, this paper primarily takes HEMUL as an example to present our design.

As we can see from the typical workflow of HEMUL in Fig. 8, HEMUL mainly involves the calculation of several polynomials in different representations (e.g., CRT and NTT representations). Actually, these polynomials are transformed from original coefficient matrices. And the size of each matrix is large and can thus be partitioned for parallel execution. Therefore, in the case of the acceleration on multiple GPUs, the opportunities come from the tremendous *data-level parallelism* provided by those large coefficient matrices, which can be exploited to map the same task on multiple GPUs to process different data partitions. Note that HE-Booster on a single GPU primarily exploits the *thread-level parallelism*, such as the local synchronization in the NTT phase. And the task running on each GPU still applies the single-GPU acceleration design.

Since the polynomials represented by coefficient matrices are usually accessed with different patterns or granularity,

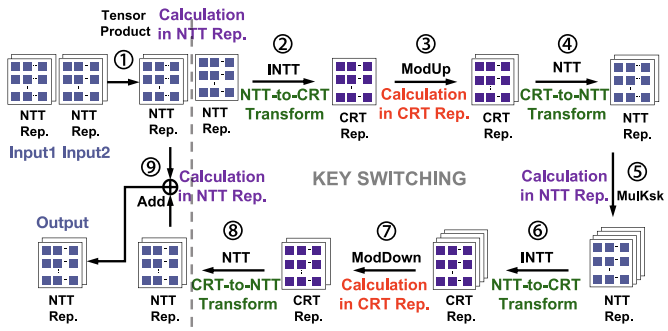

Fig. 8. A typical workflow of HEMUL.

three typical partitions on *data-level parallelism* are accordingly adopted in HE-Booster for different stages:

- *Partition at the sub-space granularity for transformation between NTT and CRT representation.*The CRT-to-NTT or NTT-to-CRT transformation of each sub-space can be processed independently. Thus, each polynomial can be partitioned into several sub-spaces, and one or multiple sub-spaces can be assigned to a GPU (Fig. 9a).
- *Partition at the segment granularity for calculation in NTT representation.*Each element is calculated (e.g., addition and multiplication) without any dependence on any other element. Therefore, each sub-space can be further divided into multiple sets consisting of several elements, and each set is named as one segment in this paper. Naturally, one or multiple segments can be mapped on a GPU (Fig. 9b).
- *Partition at the column(s) granularity for calculation in CRT representation.*Data in one or multiple columns can be independently calculated (e.g., modulus up (`ModUp`) and modulus down (`ModDown`)). Then, the set of several columns from all sub-spaces can be mapped to a single GPU (Fig. 9c).

Given those partition strategies, two important decisions need to be made. First, how many HEMUL operations can be partitioned uniformly on a single GPU. Second, how many GPUs can be employed for the acceleration of specific FHE parameters to get the balance between performance speedup and cost. They are highly dependent on the workload and the GPU hardware configurations, which are evaluated in Section 5.3.

5 Evaluation

5.1 Methodology

Software and Hardware Configuration. Our GPU server has dual-socket CPUs (Intel 10-core 3.7GHz i9-10900K CPU) and 128GB memory. For single GPU acceleration, a mainstream GPU card, NVIDIA GeForce RTX3070 GPU (5888 cores, 1.5 GHz) is selected. And for multiple GPUs acceleration, we select a high-performance server node equipped with 8 NVIDIA Tesla V100s (each GPU has 5120 cores and the frequency is 1.53 GHz). In addition, the operating system is Ubuntu 18.04.5. Our implementation is developed on CUDA Toolkit 11.2. The compiler tools include GCC 9.0.0 and CMake 3.19.3. We use the NTL library (version 11.5.3) as the arithmetic interface between CPU and GPU. NVIDIA

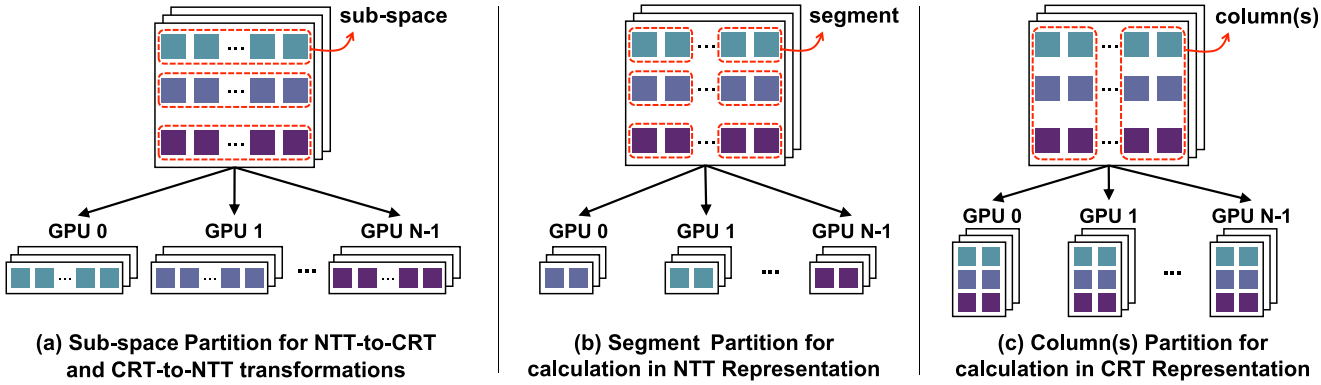

Fig. 9. Three data partition strategies on multi-GPU acceleration.

Nsight Compute tools [52] are used for profiling. We denote FHE parameter sets as (log$N$, log$Q$, L) (e.g., (13,224,4)) throughout this paper. Note that the FHE parameters in our BGV implementation provide at least 80-bit security level, which guarantees security [53].

Benchmarks. First, we evaluate the fine-grained optimizations involved in each phase of HE-Booster. Second, we conduct a breakdown of polynomial multiplication to analyze the performance gains compared to the GPU-accelerated library cuHE [24]. Third, to evaluate BGV primitives, we compare HE-Booster with three widely used CPU-based FHE libraries: HElib (version 2.2.1) [46], SEAL (version 4.0.0) [54] and PALISADE (version 1.11.2) [55]. Furthermore, we evaluate the CKKS primitives to demonstrate the scalability of HE-Booster. Besides, a detailed performance analysis of HEMUL acceleration on multiple GPUs is conducted. Finally, to further demonstrates the generalization of HE-Booster, we evaluate the performance of a typical *encrypted inference* implementation on both single-GPU and multi-GPU system.

5.2 Performance Analysis of Single-GPU Acceleration

(1) CRT/ICRT Phase Acceleration. Fig. 10 shows the performance of CRT and ICRT kernel in single-word and double-word settings. It can be observed that in both CRT and ICRT phases, using double-word CRT moduli achieves a remarkable performance improvement than single-word moduli. First, in our case, the double-word setting reduces the computation by *half* compared to the single-word. Second, while double-word arithmetic requires more transformation overhead, CUDA PTX assembly language is employed to reduce these transformations, achieving the performance benefits close to single-word arithmetic. Based on the above two points, double-word implementation can achieve a good balance between transformation overhead and computational efficiency. Therefore, it brings significant performance gains. Besides the double-word support for ICRT in cuHE, we evaluate the implementation based on *Garner's algorithm*. Experiments demonstrate that *Garner's algorithm* performs much better than cuHE, especially for large parameters.

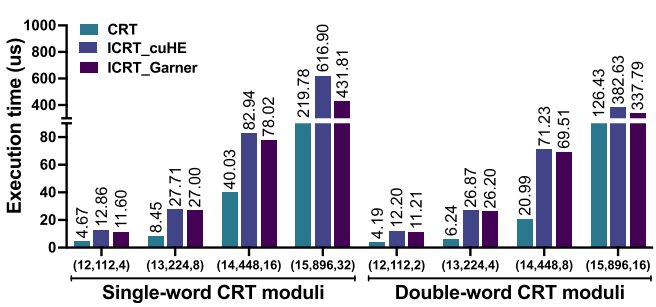

Fig. 10. Performance (microsecond) of CRT and ICRT.

(2) NTT/INTT Phase Acceleration. The performance of inter-thread local synchronization (*Local sync*), typical inter-block global synchronization, and implementations in cuHE are evaluated. We set the polynomial length, ranging from 2^{11} to 2^{16}. Corresponding performance is evaluated when the bit width of q is set to 28 and 56. Seven methods are listed below except our method:

- *CPU_sync* [29] launches kernels repeatedly;
- *Lock-based* [39] employs a global mutex;
- *Tree-based* [39] uses multiple mutexes in tree topology;
- *Lock-free sync* [39] utilizes two arrays to coordinate the synchronization requests;
- *Cooperative groups* introduced in CUDA 9 [56];
- *Hybrid method* [29] combines *CPU_sync* with block-level synchronization function *__syncthreads*().
- *cuHE* uses four-step FFT based on cyclic convolution and only only supports three lengths: 2^{13}, 2^{14}, and 2^{15}.

Fig. 11 shows the execution time of an NTT/INTT with all kinds of synchronization methods. It can be observed that our local synchronization method (*Local sync*) has the best performance. For each synchronization method, the NTT performance is similar to INTT. Thus, considering the page limits, we take NTT as an example. Compared to cuHE, *Local sync* can achieves 1.4×, 1.5×, 1.7× speedup respectively under the cases of log$N$=13, 14, 15. When log$N$=15, *Local sync* achieves 1.2× and 1.7× performance speedup than *Lock-based* and *Hybrid method* respectively under the single-word setting. Under the double-word setting, 1.3× and 1.9× speedup can be introduced. A similar performance improvement is obtained for other polynomial lengths. We remark that when log$N$=16, our GPU fails to execute 32768 threads in parallel due to resource limitations. Instead, launch 16384 threads, each of which operates with four elements instead of two, decreasing the processing efficiency. However, even in this case, our method still achieves 1.2× speedup over the *Lock-based* implementation.

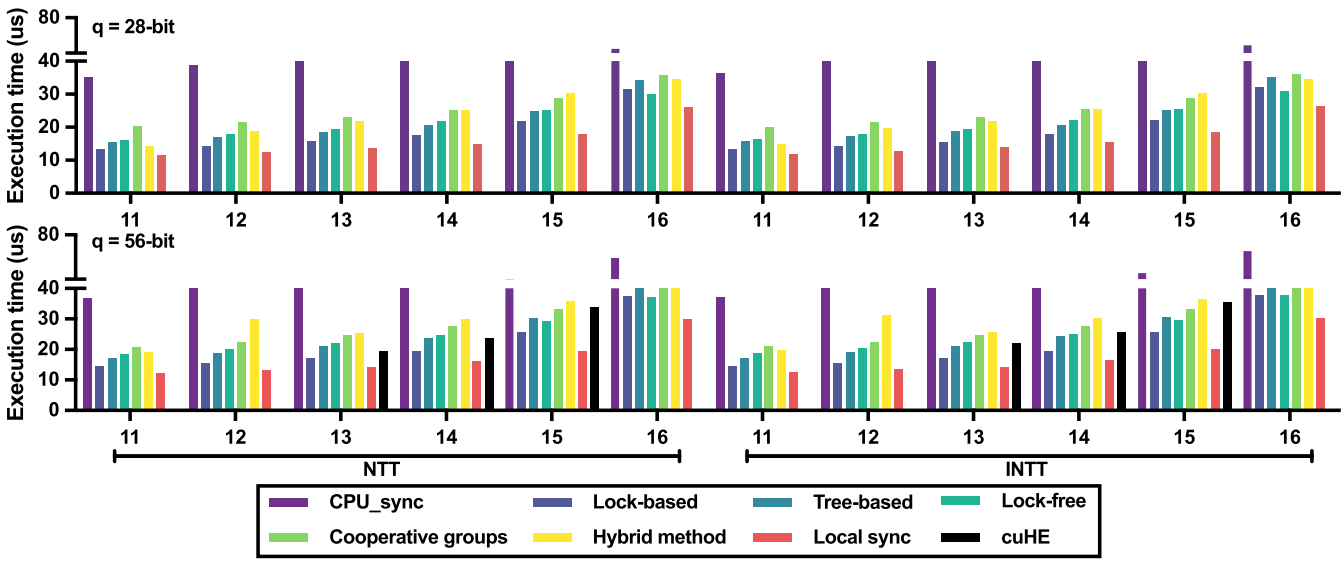

Fig. 11. Performance (microsecond) of NTT and INTT under different synchronization methods.

To better understand the waiting delay induced by GPU-based synchronization, we measure the average stalling cycles for instructions through kernel profiling with Nsight Compute tools [52]. Results shown in Fig. 12 demonstrate that *Lock-based* performs the best compared to other global synchronization. Our local synchronization (*Local sync*) induces the least latency compared to all GPU-based global synchronization.

(3) Operation Phase Acceleration. This paper conducts fine-grained experiments to evaluate the execution time of SMAC and FMAC in the key switching process. Table 1 shows the performance speedup achieved by FMAC over SMAC under different radixes. When setting radix ω to 2, SMAC incurs huge performance overheads due to massive kernel launches. With the same configuration, FMAC produces a maximum speedup of 4.3×. As the radix increases, FMAC still has more than 2.1× performance speedup. Therefore, the kernel fusion technique can greatly boost computation efficiency for key switching.

(4) Evaluation of Polynomial Multiplication in BGV scheme. Taking the representative polynomial multiplication in FHE as an example, we can better understand the acceleration effect produced by each phase of HE-Booster through fine-grained breakdown. Current state-of-the-art GPU-based cuHE is selected as the baseline. Note that cuHE (denoted as *CH*) only supports single-word implementation. Meanwhile, we implement single-word and double-word polynomial multiplication based on HE-Booster, denoted as *SW* and *DW* respectively. In addition, we use the *batch* technique, for both two settings (denoted as *SWB* and *DWB* respectively), to process polynomial vectors in NTT/INTT phase through launching only one kernel. Three kinds of parameter configurations are evaluated for each method. For instance, *DWB1* runs the first configuration with double-word *batch* NTT.

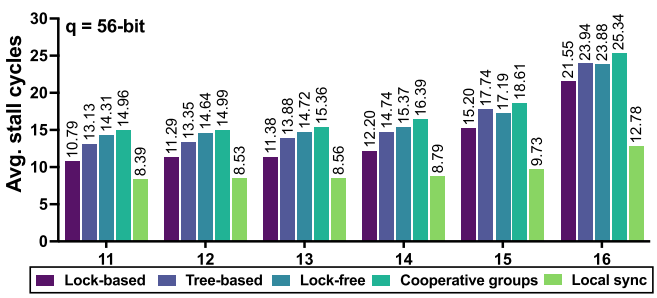

Fig. 12. Pipeline stall during kernel execution under different synchronization methods.

Fig. 13 demonstrates the performance profiling results. Three observations can be concluded: **(i)** Under the single-word setting without batch NTT (*SW*), our work achieves a 3.9×∼4.3× performance speedup compared to cuHE (*CH*). It is worth noting that in cuHE, the reduction operation occupies more than 58.1% execution time. In our implementation, negative wrapped convolution is employed to eliminate reduction and the performance is improved significantly. **(ii)** With double-word settings without batch NTT (*DW*), 1.4×∼1.7× speedup is achieved than the single-word settings without batch NTT (*SW*). The reason is that the time to perform a single NTT transform is similar for *SW* and *DW* settings, but *SW* requires almost twice NTT transforms than *DW* under the same parameters. **(iii)** Using batch NTT reduces the proportion of NTT/INTT in the total execution time and provides better performance. For instance, employing batch NTT can reduce the proportion of NTT from 84.1% to 56.7% in single-word settings. Combining all optimizations proposed in this paper, the double-word setting with batch NTT (*DWB*) achieves 9.3×∼13.3× performance speedup compared to cuHE.

TABLE 1
Performance (Microsecond) of Inner Product of the Key Switching and Speedup $\mathcal{S}$ (SMAC Versus FMAC)

logω	(13,224,4)			(14,448,8)			(15,896,16)		
	SMAC	FMAC	$\mathcal{S}$	SMAC	FMAC	$\mathcal{S}$	SMAC	FMAC	$\mathcal{S}$
1	2408.4	558.5	4.3×	10268.1	3462.1	3.0×	75609.3	29601.7	2.6×
5	483.7	130.1	3.7×	2125.8	732.2	2.9×	12643.2	5549.7	2.3×
10	247.3	78.3	3.2×	1076.4	391.1	2.8×	6321.8	2829.7	2.2×
20	129.4	48.9	2.6×	550.2	217.5	2.5×	3160.5	1459.5	2.2×
30	86.2	37.8	2.3×	358.8	151.7	2.4×	2107.2	1009.3	2.1×

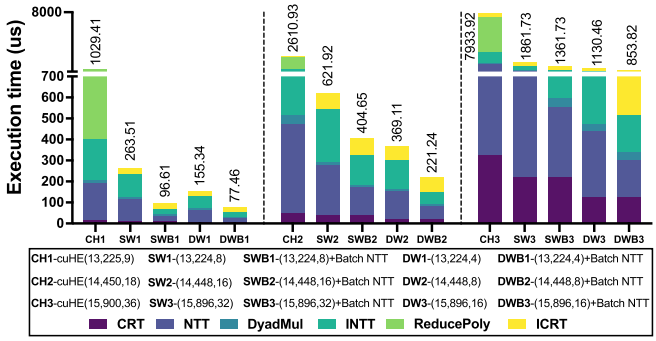

Fig. 13. Performance breakdown of polynomial multiplication in BGV under different parameter sets.

(5) Evaluation of BGV primitives. Homomorphic operations, including homomorphic addition (HEADD), homomorphic multiplication (HEMUL) and homomorphic rotation (HEROT) are fundamental primitives for computing on packed ciphertext, and their performance reflects the practicality of the implementation of FHE schemes. Built on the HE-Booster, we implement the BGV scheme on the GPU (denoted as BGVGPU) and make a comparison with three widely used CPU-based libraries HElib, SEAL, and PALISADE. We also implement the BGV scheme based on cuHE (single-word setting). Note that since the execution time of encryption and decryption is less, it is not considered for acceleration by default. Furthermore, both HEMUL and HEROT implementations involves the *key switching* operation. The performance results are shown in Table 2.

First, for performance-critical HEMUL, HE-Booster achieves the performance speedup of up to 251.7×, 78.5× and 164.9× compared to HElib, SEAL, and PALISADE, respectively. Second, for HEADD with low-performance requirements, HE-Booster can bring up to 9.2×, 50.6× and 91.2× speedup compared to HElib, SEAL, and PALISADE. Third, our implementation achieves 27.5×, 101.3×, and 170.5× speedup for HEMUL and 4.0×, 6.0× and 4.4× speedup for HEADD over cuHE. It is worth noting that the cuHE-based BGV implementation does not show significant performance boosts, and is even slower than the CPU implementations. It is mainly because the NTT implementation in cuHE needs to perform a polynomial reduction operation, causing a serious performance bottleneck (see Section 5.2(4)). Also, the cuHE does not employ the *batch* NTT method to maximize parallelism. Furthermore, we evaluate the performance of HEROT. Note that PALISADE does not provide HEROT benchmark for the BGV scheme, thus we do not consider it here. Table 3 shows that HE-Booster produces up to 869.7×, 63.9× and 177.5× performance gains compared to HElib, SEAL, and cuHE.

(6) Evaluation of CKKS primitives. We further extend to the CKKS scheme (denoted as CKKSGPU) to demonstrate the performance gains built on the HE-Booster. For comparison, we use three CPU-based libraries (i.e., SEAL, HElib, and PALISADE) as the baseline. cuHE is not considered due to its poor performance in HEMUL and HEROT. Note that we extend PALISADE's benchmark to support the performance evaluation of CKKS. Three homomorphic operations, including HEADD, HEMUL, and HEROT are evaluated. The performance results are shown in Table 4.

- *The speedup of HEADD.* HE-Booster produces similar performance gains (e.g., 9.0× over HElib, 54.0× over SEAL, and 91.8× over PALISADE) for accelerating HEADD of BGV and CKKS. This is because theoretically BGV and CKKS have the same ciphertext size and execution time under the same parameters (polynomial length N, ciphertext modulo Q).
- *The speedup of HEMUL.* Although CKKS scheme implemented by all three CPU software libraries performs much better than the BGV scheme, our implementation brings 42.5×, 67.3× and 47.5× speedup for HElib, SEAL, and PALISADE, respectively.
- *The Speedup of HEROT.* HE-Booster produces up to 45.3×, 66.7×, and 51.9× performance improvements over HElib, SEAL, and PALISADE, which also involve key switching operation and have comparable performance to HEMUL.

Besides, we evaluate the *ReScale* operation, which can divide a plaintext by an integer to remove some inaccurate LSBs as a rounding step in usual approximate computations. We remark that HElib does not provide *ReScale* evaluation in the CKKS benchmark, so it is not considered here. Table 5 shows that HE-Booster produces 40.6× and 29.5× performance boosts for SEAL and PALISADE, respectively.

5.3 Performance Analysis of Accelerating HEMUL on Multiple GPUs

As shown in Fig. 14, we evaluate the execution time of performing 1, 2, 4, and 8 BGV HEMULs with three configurations (FHE parameters) on different numbers of GPUs (NVIDIA Telsa V100). Overall, HE-Booster exhibits good scalability on multiple GPUs. For example, under the FHE parameter of (N=32768,L=16), mapping 8 HEMULs to 8 GPUs can achieve more than 7.66× performance speedup compared to a single GPU. Specific analysis is detailed as follows.

TABLE 2
Performance (Millisecond) of BGV Homomorphic Addition and Homomorphic Multiplication and Speedup $\mathcal{S}_1$ (HElib Versus BGVGPU), $\mathcal{S}_2$ (SEAL Versus BGVGPU), $\mathcal{S}_3$ (PALISADE Versus BGVGPU) and $\mathcal{S}_4$ (cuHE Versus BGVGPU)

log$N$	log$Q$	Homomorphic Addition									Homomorphic Multiplication								
		HElib	SEAL	PALISADE	cuHE	BGVGPU	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_4$	HElib	SEAL	PALISADE	cuHE	BGVGPU	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_4$
13	218	0.03	0.19	0.62	0.04	0.01	3.0×	19.0×	62.0×	4.0×	35.90	6.85	7.17	7.72	0.28	128.2×	24.5×	25.6×	27.5×
14	438	0.11	0.81	1.11	0.12	0.02	5.5×	40.5×	55.5×	6.0×	125.50	35.75	57.14	69.9	0.69	181.9×	51.8×	82.8×	101.3×
15	881	0.58	3.19	5.47	0.264	0.06	9.7×	53.2×	91.2×	4.4×	609.00	189.86	399.01	412.50	2.42	251.7×	78.5×	164.9×	170.5×

BGVGPU), S_2 (SEAL vs. BGVGPU), S_3 (PALISADE vs. BGVGPU) and S_4 (cuHE vs. BGVGPU).

TABLE 3
Performance (Millisecond) of BGV Homomorphic Rotation and Speedup $\mathcal{S}_1$ (HElib Versus BGVGPU), $\mathcal{S}_2$ (SEAL Versus BGVGPU) and $\mathcal{S}_3$ (cuHE Versus BGVGPU)

$\log N$	$\log Q$	Homomorphic Rotation						
		HElib	SEAL	cuHE	BGVGPU	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$
13	218	89.10	4.22	6.65	0.23	387.4×	18.4×	28.9×
14	438	307.00	24.21	64.89	0.61	503.3 ×	39.7×	106.4×
15	881	2009.00	147.58	409.97	2.31	869.7 ×	63.9×	177.5×

- Because a single HEMUL with small FHE parameters has less computation workload, partitioning multiple HEMULs together can improve GPU utilization. For instance, under the FHE parameter of (N=8192,L=4), executing 2 HEMULs sequentially on a single GPU takes 424us (2×212us). When divided uniformly, most of the polynomials from the two HEMULs can be computed in parallel and consume only 334us.
- Since GPU computation resources are limited, processing too much data in a kernel results in worse performance. Table 6 shows the execution time of the most computation-intensive CRT-to-NTT transform for executing 4 HEMULs while partitioning the different numbers of HEMULs uniformly. Under the FHE parameter of (N=16384,L=8), partitioning 4 HEMULs requires 413us, which is much longer than partitioning 2 HEMULs uniformly (250us).
- When the computation becomes intensive (e.g., increasing the number of HEMULs or using larger FHE parameters), mapping them into multiple GPUs can achieve significant speedup. Take the case of the parameter (N=16384,L=8) as an example, 5.07× speedup is observed when accelerating 1 MULs on 8 GPU. Accordingly, 7.19× speedup can be achieved while mapping 8 HEMULs. Similar results can be observed under other FHE parameters as well.
- Under the configuration of (N=8192,L=4), executing 8 HEMULs on 8 GPUs only introduces 1.57× performance improvement compared to execution on 4 GPUs. The reason is that the most time-consuming NTT and INTT can only be partitioned at the granularity of sub-spaces. That is to say, NTT and INTT can only be assigned to 4 GPUs, and the performance improvement will be less if more GPUs are used. In a real-life application, the computation workloads can be profiled to determine the number of GPUs to use for a better balance between performance and cost.

5.4 Performance Analysis of an End-to-End Application

With HE-Booster, we present a BGV-based *encrypted inference* implementation as an end-to-end application, which generally takes server's plaintext models (e.g., unencrypted weights) and client's ciphertext data. Specifically, a typical five-layer CNN structure is used in many studies[7], [8], [9], [10], which consists of:

- *Convolution layer*: This layer has five 5×5 filter kernels and stride size of 2. A zero padding is added to the original image. Therefore, the output size is 5×13×13.
- *Activation layer*: The *square* function is utilized to approximate the nonlinear *ReLU* function.
- *Dense (fully connected) layer*: This layer is fully connected with 845 input nodes and 100 output nodes, which means multiplying by a matrix of size 845×100.
- *Activation layer*: This layer still squares the value of each node, like step 2).
- *Dense (fully connected) layer*: This layer is fully connected with 100 input nodes and 10 output nodes, which means multiplying by a matrix of size 100×10.

Besides, we train the approximated model using PyTorch library[57] on the MNIST dataset [58], which consists of 60,000 images (50,000 images for train and 10,000 images for test) of hand written digits. Each image consists of 28×28 pixels, which is represented by its gray value between 0 and 255. Note that ciphertext *batching* usually brings a decent amortized per instance time, but it makes no difference if 1 or 8192 instances are being processed (e.g., same number of ciphertext). Therefore, in this case, we directly encrypt a single MNIST image, resulting in 28×28 = 784 ciphertexts. The encryption scheme is initialized on the client-side. Like HCNN[10], we set the plaintext modulus to a 43-bit prime number and the ciphertext modulus to a 330-bit prime number. In this way, the message size transferred from client to server is 505.3 MB. The output of the network transferred from server to client consists of 10 outputs of the final fully connected layer, which has a size of 6.4 MB.

Table 7 lists the number of the homomorphic operations (HOPs) of *encrypted inference* over a single encrypted image. In particular, CT-CT Adds and CT-CT Mults denote HEADD and HEMUL operation in BGV evaluation (as shown in Table 2), respectively. **(i)** For single-GPU acceleration: Table 8 lists five typical *encrypted inference* implementations. In particular, we use CryptoDL as the baseline due to the same BGV scheme adopted. Meanwhile, we also list recent implementations based on other FHE schemes. Specifically, HE-Booster achieves 61.0×, 76.2×, 9.3× and 2.21×

TABLE 4
Performance (Millisecond) of CKKS Homomorphic Addition, Homomorphic Multiplication, Homomorphic Rotation and Speedup $\mathcal{S}_1$ (HElib Versus CKKSGPU), $\mathcal{S}_2$ (SEAL Versus CKKSGPU) and $\mathcal{S}_3$ (PALISADE (Denoted as PALI.) Versus CKKSGPU)

$\log N$	$\log Q$	Homomorphic Addition							Homomorphic Multiplication							Homomorphic Rotation						
		HElib	SEAL	PALI.	CKKSGPU	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	HElib	SEAL	PALI.	CKKSGPU	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	HElib	SEAL	PALI.	CKKSGPU	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$
13	218	0.04	0.20	0.58	0.01	4.0×	20.0×	58.0×	5.89	4.48	6.98	0.21	28.0×	21.3×	33.2×	5.42	3.64	7.19	0.18	30.1×	20.2×	39.9×
14	438	0.14	0.82	1.09	0.02	7.0×	41.0×	54.5×	19.8	26.04	22.74	0.58	34.1×	44.9×	39.2×	18.71	22.43	23.75	0.47	39.8×	47.7×	50.5×
15	881	0.54	3.24	5.51	0.06	9.0×	54.0×	91.8×	94.32	149.41	98.38	2.22	42.5×	67.3×	44.3×	92.43	136.12	105.78	2.04	45.3×	66.7×	51.9×

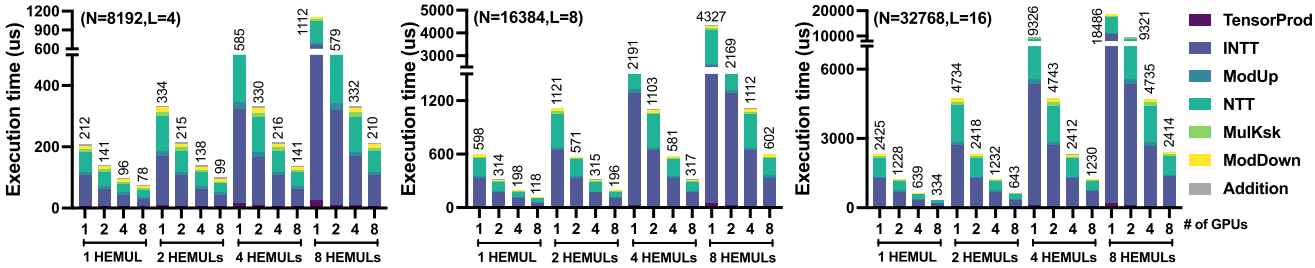

Fig. 14. Performance evaluation of homomorphic multiplication on multiple GPUs.

speedup compared to CryptoNets[7], CryptoDL[8], Faster CryptoNets (FCryptoNets)[9] and SHE [11]. It should be noted that the speedup achieved on NN inference is slightly smaller than HEADD and HEMUL of the BGV evaluation. We conclude the following reasons.

- *Different FHE parameter setting.* The CPU-based implementations employ smaller parameters (e.g., smaller ciphertext modulus) and thus have much less ciphertext, involving less time-consuming homomorphic operations (e.g., HEMults).
- *FHE scheme they rely on.* SHE[11] uses the bit-wise TFHE scheme and quantifies the input data and model weights to 0-1 bits. Similarly, CryptoNets[7] uses the YASHE scheme[59], whose ciphertext contains only one polynomial instead of a polynomial pair. Thus, they significantly reduce the ciphertext size and the computational complexity.
- *Specialized model optimizations.* Faster CryptoNets[9] utilizes *pruning* strategy to reduce the number of model weights and the number of homomorphic operations (e.g., almost 1/10 of the original model).

Compared to the state-of-the-art GPU implementation HCNN[10], HE-Booster also achieves a 1.23× performance boost. The performance improvement over HCNN derives from two major aspects. First, the BGV scheme used in HE-Booster provides a faster homomorphic multiplication (e.g., CT-CT Mults here) than the BFV scheme adopted in HCNN [32]. Second, our implementation involves fewer time-consuming CT-CT Mults than HCNN. **(ii)** For multi-GPU acceleration: We further evaluate the performance of the multi-GPU system. When using 2, 4, and 8 GPU cards, the time of one encrypted inference is 2.8s, 2.1s, 1.2s, and the corresponding speedup is 1.54×, 2.04×, and 3.58×.

6 Related Work

Recently, utilizing hardware accelerators to improve FHE performance has become of interest for researchers. In the following, we review FHE acceleration implementation for different hardware platforms.

GPU-Based FHE Acceleration. Wang et al. [60] propose the first GPU acceleration method that targets Gentry-Halevi [61] scheme. They present another work [62] later to accelerate the learning-with-error (LWE) version of the BGV scheme. However, they only provide optimization with the CRT algorithm, instead of the complete procedure. Dong et al. [63] utilize cuFFT library [64] to accelerate the BGV encryption primitive in HElib library. Badawi et al. [25][49] successively accelerate the BEHZ[65] and HPS[66] variants of the original BFV scheme. We remark that their works also adopt single-word CRT moduli. The difference is that NTT is replaced by *Discrete Galois Transform* (DGT), which requires additional pre-process and post-process on vectors. Moreover, Badawi et al. [10] propose HCNN that implements BFV-based *encrypted inference* on a single GPU. Our experiments demonstrate that HE-Booster has better performance than HCNN. Furthermore, they present a multi-GPU design[67] that implements the HPS variant of the BFV scheme, in which complicated homomorphic multiplication process demands more computation overhead. Compared to their work, we propose three fine-grained data partitioning strategies for different stages in typical HEMUL workflow to achieve better scalability. For the CKKS scheme, Jung et al. [26] utilize two kernels to accelerate NTT/INTT. In the other work [68], they extend the acceleration to Residue Number System (RNS) CKKS scheme[34], [69] with large parameters. cuHE [24] is the current state-of-the-art GPU-based implementation to accelerate polynomial arithmetic. But its single-word CRT/ICRT is not adaptive, and cyclic convolution-based NTT/INTT introduces performance overheads. In our work, these issues are addressed, and the experiments demonstrate that 8.0×∼12.1× performance speedup is achieved compared to cuHE. Besides, Shen et al. [70] propose CARM, targeting for accelerating the BGV, BFV and CKKS schemes using GPU. However, they fail to evaluate the complete homomorphic multiplication

TABLE 5
Performance (Millisecond) of CKKS *ReScale* Operations and Speedup $\mathcal{S}_1$ (SEAL Versus CKKSGPU) and $\mathcal{S}_2$ (PALISADE Versus CKKSGPU)

log$N$	log$Q$	ReScale				
		SEAL	PALISADE	CKKSGPU	$\mathcal{S}_1$	$\mathcal{S}_2$
13	218	1.02	1.18	0.06	17.0×	19.7×
14	438	4.51	2.78	0.13	34.7×	21.4×
15	881	18.29	13.26	0.45	40.6×	29.5×

TABLE 6
Execution Time (Microsecond) of CRT-to-NTT Transforms on a Single GPU While Performing 4 HEMULs Through Uniform Partition in Different Numbers of HEMULs

log$N$	log$Q$	# HEMULs partitioned uniformly		
		1	2	4
13	218	108	82	73
14	438	288	250	413
15	881	882	958	956

TABLE 7
Breakdown of Homomorphic Operations (HOPs) for Each Layer, Where PT Denotes Plaintext and CT Denotes Ciphertext

NN Layer	HOPs	PT-CT Adds	CT-CT Adds	PT-CT Mults	CT-CT Mults
Conv-1	43095	845	20280	21125	–
Act-1	845	–	–	–	845
Dense-1	169100	100	84400	84500	–
Act-2	100	–	–	–	100
Dense-2	2010	10	990	1000	–
Total HOPs	215150	955	105670	106625	945

Adds *and* Mults *refer to the number of homomorphic addition and multiplication. Dashes indicate zero operations.*

TABLE 8
Comparison of Different *Encrypted Inference* Implementation

Criteria	CryptoNets	CryptoDL	FCryptoNets	HCNN	SHE	HE-Booster
Time (second)	250	320	39.1	5.16	9.3	4.2
Accuracy (%)	98.95	98.52	98.71	99.00	99.54	98.96
Memory (MB)	367.5	336.7	411.1	505.3	123	505.3
FHE Scheme	YASHE	BGV	BFV	BFV	TFHE	BGV

primitive, especially the performance-dominant key-switching operation involved. Finally, targeting on Fully Homomorphic Encryption over Torus (TFHE) [71], two GPU FHE libraries, cuFHE [72] and NuFHE [73], are released. Morshed et al. [74] proposed new optimizations to achieve further speedup, and the code has been open source.

FPGA-Based FHE Acceleration. Cousins et al. [75] propose the first FPGA-based acceleration architecture without real implementation. Roy et al. [17] implement the BFV scheme with a polynomial degree of 2^{15}. But there is not a significant performance speedup because of the off-chip data transfer limitation. In other work [76], they optimize the BFV scheme by reducing the degree of the polynomial to 2^{12}. HEAX [77] proposes a more optimized architecture targeted at CKKS schemes. Su et al. [78] and Paul et al. [79] present FPGA-based RLWE accelerators targeted on BGV scheme. FAB [80] presents a multi-FPGA FHE acceleration system to boost performance, especially to accelerate bootstrapping in CKKS. HEAWS [81] presents the implementation on the Amazon AWS FPGA, which can perform 613 homomorphic multiplications per second for a parameter set. Existing schemes use fixed-size parameter sets [16], [17], [18] that limit the adaptive FHE configurations in diverse application scenarios. Moreover, such FPGA-based solutions face the challenge of productivity.

ASIC-Based FHE Acceleration. Recently, F1 [19], CraterLake[20], BTS[21], ARK[22], and BASALISC[23] respectively design hardware accelerator architectures that target different optimizations (e.g., programmability for flexible parameter settings, bootstrapping for unbounded multiplicative depth or memory access for low-overhead data movement), but they have only been implemented on simulators so far. Although these accelerators demonstrate promising speedup according to the simulation results, it is still important to study the design of GPU-based FHE acceleration. First, one fact is that GPUs have been widely deployed in commercial data centers, it is also known that ASIC development is very expensive and deploying a new ASIC chip also has an additional purchase cost. Second, the major innovations of HE-Booster can also be employed in the hardware accelerator architectures. Specifically, the local-synchronization method can be used to accelerate NTT/INTT in hardware architectures.

7 Conclusion

This paper presents an efficient GPU-based polynomial arithmetic acceleration called HE-Booster. Focusing on the five common phases in typical FHE schemes, HE-Booster presents a systematic acceleration. In the case of single GPU acceleration, we observe the regularity of stage-to-stage processing for NTT and INTT phases and propose the local synchronization to allow threads to be executed as early as possible. Furthermore, exploiting the tremendous *data-level parallelism*, we propose a scalable partition strategy for multiple GPUs. Experiments on a single NVIDIA GPU platform demonstrate that HE-Booster outperforms 251.7×, 78.5× and 164.9× than three mainstream CPU-based libraries HElib, SEAL, and PALISADE, and 27.5×∼170.5× speedup is obtained compared to the GPU-accelerated cuHE. Experiments on Tesla V100s show that HE-Booster exhibits good scalability on multiple GPUs. Mapping 8 HEMULs to 8 GPUs can achieve more than 7.66× performance speedup compared to a single GPU.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] T. Oladunni and S. Sharma, "Homomorphic encryption and data security in the cloud," in *Proc. 28th Int. Conf.*, vol. 64, 2019, pp. 129–138.
[2] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *J. Netw. Comput. Appl.*, vol. 34, no. 1, pp. 1–11, 2011.
[3] C. Rong, S. T. Nguyen, and M. G. Jaatun, "Beyond lightning: A survey on security challenges in cloud computing," *Comput. Elect. Eng.*, vol. 39, no. 1, pp. 47–54, 2013.

[4] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 1999, pp. 223–238.

[5] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[6] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. 31, no. 4, pp. 469–472, Jul. 1985.

[7] 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.*, 2016, pp. 201–210.

[8] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Deep neural networks over encrypted data," 2017, *arXiv:1711.05189*.

[9] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster CryptoNets: Leveraging sparsity for real-world encrypted inference," 2018, *arXiv:1811.09953*.

[10] A. Al Badawi et al., "Towards the AlexNet moment for homomorphic encryption: HCNN, thefirst homomorphic CNN on encrypted data with GPUs," 2018, *arXiv:1811.00778*.

[11] Q. Lou and L. Jiang, "SHE: A fast and accurate deep neural network for encrypted data," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 10035–10043.

[12] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Proc. Annu. Cryptol. Conf.*, 2012, pp. 868–886.

[13] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Proc. Annu. Cryptol. Conf.*, 2011, pp. 505–524.

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

[15] 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.*, 2017, pp. 409–437.

[16] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 143–163.

[17] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.

[18] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 387–398.

[19] N. Samardzic et al., "F1: A fast and programmable accelerator for fully homomorphic encryption," in *Proc. IEEE/ACM 54th Annu. Int. Symp. Microarchit.*, 2021, pp. 238–252.

[20] N. Samardzic et al., "CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2022, pp. 173–187.

[21] S. Kim et al., "BTS: An accelerator for bootstrappable fully homomorphic encryption," 2021, *arXiv:2112.15479*.

[22] J. Kim et al., "ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," 2022, *arXiv:2205.00922*.

[23] R. Geelen et al., "BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption," 2022, *arXiv:2205.14017*.

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

[25] 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, pp. 70–95, 2018.

[26] W. Jung et al., "Heaan demystified: Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," 2020, *arXiv:2003.04510*.

[27] Z. Zheng, "Encrypted cloud using GPUs," PhD dissertation, Master's Thesis, KU Leuven, Leuven, Belgium, 2020.

[28] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[29] O. Ozerk, C. Elgezen, A. C. Mert, E. Oztürk, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," *J. Supercomput.*, vol. 78, pp. 2840–2872, 2022.

[30] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2010, pp. 1–23.

[31] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Proc. Annu. Cryptol. Conf.*, 2012, pp. 850–867.

[32] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," *IACR Cryptol. ePrint Arch.*, vol. 2021, 2021, Art. no. 204.

[33] S. Halevi and V. Shoup, "Design and implementation of HElib: A homomorphic encryption library," *Cryptol. ePrint Arch.*, pp. 1–42, 2020.

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

[35] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2012, pp. 465–482.

[36] J. M. Pollard, "The fast fourier transform in a finite field," *Math. Comput.*, vol. 25, no. 114, pp. 365–374, 1971.

[37] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 698–706, Mar. 2015.

[38] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2014, pp. 371–391.

[39] S. Xiao and W.-C. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.

[40] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers," in *Proc. Int. Conf. cryptol. Inf. Secur. Latin Amer.*, 2015, pp. 346–365.

[41] V. Volkov, "Better performance at lower occupancy," in *Proc. GPU Technol. Conf.*, 2010, Art. no. 16.

[42] A. Li, W. Liu, L. Wang, K. Barker, and S. L. Song, "Warp-consolidation: A novel execution model for GPUs," in *Proc. Int. Conf. Supercomput.*, 2018, pp. 53–64.

[43] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2015, pp. 12–22.

[44] 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.*, 1986, pp. 311–323.

[45] Garner and L. Harvey, "The residue number system," *Ire Trans. Electron. Comput.*, vol. EC-8, no. 2, pp. 140–147, 1959.

[46] S. Halevi and V. Shoup, "Algorithms in HElib," in *Proc. Annu. Cryptol. Conf.*, 2014, pp. 554–571.

[47] C. Hao, K. Laine, and R. Player, "Simple encrypted arithmetic library - seal V2.1," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2017, pp. 3–18.

[48] W. Jung et al., "Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *IEEE Access*, vol. 9, pp. 98 772–98 789, 2021.

[49] A. Q. 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, Apr.–Jun. 2021.

[50] N. Compute, "PTX: Parallel thread execution ISA version 2.3," *Dostopno na*: http://developer.download.nvidia.com/compute/cuda, vol. 3, pp. 1–203, 2010.

[51] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. Int. Conf. Cyber, Phys. Social Comput.*, 2010, pp. 344–350.

[52] "Nsight compute roofline analysis," 2022. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline

[53] M. Albrecht et al., "Homomorphic encryption security standard," HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., Nov. 2018. [Online]. Available: https://homomorphicencryption.org/

[54] Microsoft SEAL (release 4.0). Mar. 2022. [Online]. Available: https://github.com/Microsoft/SEAL

[55] Y. Polyakov, K. Rohloff, and G. W. Ryan, "Palisade lattice cryptography library user manual," Cybersecurity Res. Center, New Jersey Inst. Technol., Tech. Rep., vol. 15, 2017. [Online]. Available: https://palisade-crypto.org
[56] M. Harris and K. Perelygin, "Cooperative groups: Flexible CUDA thread programming," NVIDIA Developer Blog, 2017. [Online]. Available: https://palisade-crypto.org
[57] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[58] Y. LeCun, "The MNIST database of handwritten digits," 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[59] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *Proc. IMA Int. Conf. Cryptogr. Coding*, 2013, pp. 45–64.
[60] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, 2012, pp. 1–5.
[61] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2011, pp. 129–148.
[62] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using GPU," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2014, pp. 2800–2803.
[63] J. Dong, "Accelerating BGV scheme of fully homomorphic encryption using GPUs," Ph.D. dissertation, Worcester Polytechnic Institute, Worcester, MA, USA, 2016.
[64] Cufft. [Online]. Available: https://docs.nvidia.com/cuda/archive/8.0/cufft/index.html
[65] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2016, pp. 423–442.
[66] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Proc. Cryptographers' Track RSA Conf.*, 2019, pp. 83–105.
[67] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. K. M. Mi, "Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 379–391, Feb. 2021.
[68] 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, 2021, Art. no. 508.
[69] 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.*, 2018, pp. 347–368.
[70] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "CUDA-accelerated RNS multiplication in word-wise homomorphic encryption schemes," *Cryptol. ePrint Arch.*, pp. 1–19, 2022.
[71] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2016, pp. 3–33.
[72] W. Dai, "CUDA-accelerated fully homomorphic encryption library," 2018. [Online]. Available: https://github.com/vernamlab/cuFHE
[73] B. Opanchuk, "NuFHE, a GPU-powered torus fhe implementation," 2020. [Online]. Available: https://github.com/nucypher/nufhe
[74] T. Morshed, M. M. Al Aziz, and N. Mohammed, "CPU and GPU accelerated fully homomorphic encryption," in *Proc. IEEE Int. Symp. Hardware Oriented Secur. Trust*, 2020, pp. 142–153.
[75] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of homomorphic encryption," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.
[76] S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 387–398.
[77] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 1295–1309.
[78] Y. Su, B. Yang, C. Yang, and L. Tian, "FPGA-based hardware accelerator for leveled ring-lwe fully homomorphic encryption," *IEEE Access*, vol. 8, pp. 168 008–168 025, 2020.
[79] B. Paul, T. K. Yadav, B. Singh, S. Krishnaswamy, and G. Trivedi, "A resource efficient software-hardware co-design of lattice-based homomorphic encryption scheme on the FPGA," *IEEE Trans. Comput.*, pp. 1–14, 2022, doi: 10.1109/TC.2022.3198628.
[80] R. Agrawal et al., "FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption," 2022, *arXiv:2207.11872*.
[81] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the amazon AWS FPGA," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020.

Zhiwei Wang received the BE degree from the College of Engineering, Qufu Normal University in 2016 and the ME degree from the Automation Research and Design Institute of Metallurgical Industry, co-educated with Chinese Academy of Sciences in 2019. He is currently working toward the PhD degree with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. His current research interests include homomorphic encryption and hardware acceleration.

Peinan Li received the BE degree from the School of Computer Information and Technology, Shanxi University in 2014, the ME degree in computer technology from the Harbin University of Science and Technology, co-educated with the Institute of Automation, Chinese Academy of Sciences in 2017, and the PhD degree from Institute of Information Engineering, Chinese Academy of Sciences, in 2021. He is currently an assistant professor with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. His current research interests include computer architecture and hardware security.

Rui Hou received the BS and MS degrees in computer architecture from the Harbin Institute of Technology, China, in 2001 and in 2003 respectively, and the PhD degree in computer architecture from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2007. He is a professor and vice director with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences. He published more than 40 papers in international conferences and journals, and got more than 50 patents. His current research interests include computer architecture, processor security, data center server architecture and AI security.

Zhihao Li received the BE degree from the College of Mathematics and Science, Jinzhong University in 2017 and the ME degree in computer information and technology, Shanxi University in 2020. He is currently working toward the PhD degree with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. His current research interests include homomorphic encryption and cryptography.

Jiangfeng Cao received the BE degree from the Anhui Normal University in 2014 and the ME degree in Computer Technology from the University of Chinese Academy of Sciences in 2017. He is a doctoral candidate with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing. His current research interests include computer architecture and AI security.

XiaoFeng Wang received the PhD degree in computer engineering from Carnegie Mellon University in 2004. Since 2010, he has been the acting director of the Security Informatics Program with Indiana University Bloomington, where he is currently a professor with the School of Informatics. His research interests include cloud and mobile security, and data and health informatics security.

Dan Meng received the BS, MS, and PhD degrees from the Harbin Institute of Technology, Harbin, Heilongjiang, China. He is the director with the Institute of Information Engineering, Chinese Academy of Sciences, and the dean with the School of Cyber Security, University of Chinese Academy of Science, Beijing, China. His research interests include high performance computer architecture and cyber security.