CARM: CUDA-Accelerated RNS Multiplication in Word-Wise Homomorphic Encryption Schemes for Internet of Things

Shiyu Shen, Hao Yang, Yu Liu, Zhe Liu, and Yunlei Zhao

Abstract—Homomorphic encryption (HE), which allows computation over encrypted data, has often been used to preserve privacy. However, the computationally heavy nature and complexity of network topologies make the deployment of HE schemes in the Internet of Things (IoT) scenario difficult. In this work, we propose CARM, the first optimized GPU implementation that covers BGV, BFV and CKKS, targeting for accelerating homomorphic multiplication using GPU in heterogeneous IoT systems. Our solution is suitable for accelerating RNS homomorphic multiplication on both high-performance and embedded GPUs, as it is a parametric and generic design and offers various trade-offs between resource and efficiency. We offer constant-time low-level arithmetic with minimum instructions and memory usage, as well as performance- and memory-prior configurations. Through this, we can provide more real-time evaluation results and relieve the computational pressure on cloud devices. We deploy our implementations on two GPUs. Compared to the CPU implementation, we achieve up to $378.4\times$, $234.5\times$, and $287.2\times$ speedup for homomorphic multiplication of BGV, BFV, and CKKS on Tesla V100S, and $8.8\times$, $9.2\times$, and $10.3\times$ on Jetson AGX Xavier, respectively.

Index Terms—GPU acceleration, Homomorphic encryption, Internet of Things, number theoretic transform, RNS multiplication

1 Introduction

The Internet of Things (IoT) is now infiltrating into all parts of people's lives. The resource-constrained nature of IoT edge devices makes it necessary for cloud servers to assist in processing data in some scenarios, which leads to security issues when the data is privacy-sensitive. In detail, processing user data in plaintext form will leak personal information that could be used to track user interests, locations, and so on. Homomorphic encryption (HE) is often introduced to solve this problem, which allows data analysis to be performed on ciphertext and enables secure device-to-cloud computation.

Since Gentry proposed the first fully homomorphic encryption (FHE) scheme [1], [2], there have been many improvements to make FHE more practical, such as enabling Single Instruction Multiple Data (SIMD) operations [3] and handling floating point data [4]. In view of the good parallelism of the low-level arithmetic of HE schemes, parallel computing is usually used in practice to speed up the computing process, relieving the computational overhead of the schemes. The Graphics Processing Unit (GPU) has powerful parallel computing capability. Existing approaches that take full advantage of GPUs to accelerate compute-intensive computation achieve auspicious speedup, underlining their tremendous impact.

However, the computational overhead is a significant consideration that limits the deployability of homomorphic encryption, especially on resource-constrained IoT edge devices. Additionally, the complexity of IoT topologies and the diversity of the computational power of devices make the implementation more difficult. Fortunately, the massive increase in the quantities and varieties of devices has promoted the evolution of IoT topologies, making the modern IoT systems become more heterogeneous. The emergence of devices with embedded GPUs brings new challenges and opportunities to the deployment of homomorphic encryption in IoT scenarios. Embedded GPUs are a promising technology, as they provide high performance computing platform for massive data processing with low energy consumption, while preserving the portable and mobile nature of IoT devices. Enabling homomorphic encryption on such devices allows computation in the entire device-to-cloud scenario to be accelerated using GPUs, which is of great value for the deployment of FHE schemes in the IoT systems.

- *Shiyu Shen, Yu Liu, and Yunlei Zhao are with the School of Computer Science, Fudan University, Shanghai 200437, China, and also with the State Key Laboratory of Cryptology, Beijing 100878, China. E-mail: {shenshiyu21, yu_liu21}@m.fudan.edu.cn, ylzhao@fudan.edu.cn.*
- *Hao Yang is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210095, China. E-mail: crypto@d4rk.dev.*
- *Zhe Liu is with the Zhejiang Lab, Hangzhou, Zhejiang 311121, China, and the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210095, China. E-mail: zhe.liu@nuaa.edu.cn.*

Manuscript received 21 May 2022; revised 26 November 2022; accepted 3 December 2022. Date of publication 8 December 2022; date of current version 9 June 2023.
This work was supported in part by the National Key R&D Program of China under Grant 2022YFB2702000, in part by the National Natural Science Foundation of China under Grants 62132008, 62002239, and 62071222, in part by the Natural Science Foundation of Jiangsu Province, China under Grant BK20220075, and in part by the National Key R&D Program of China under Grant 2022YFB2701600.
(Corresponding Authors: Zhe Liu and Yunlei Zhao.)
Recommended for acceptance by J. Hormigo.
Digital Object Identifier no. 10.1109/TC.2022.3227874

1.1 Related Work

In 2009, Gentry proposed the first FHE scheme based on ideals [1], [2]. Since then, research in this area began to burgeon, mainly focusing on improving the efficiency and applicability of FHE schemes. Modern FHE schemes are based on different hardness assumptions, i.e., Approximate Greatest Common Divisor (AGCD) [5], (Ring-)Learning with Errors ((R-)LWE) [6], [7], and NTRU [8], and most of them can be classified into bit-wise FHE (e.g., FHEW [9] and TFHE [10]) and word-wise FHE (e.g., BGV [11], BFV [12] and CKKS [4]). Compared to bit-wise FHE, word-wise FHE is more efficient in large vectorial arithmetic operations. Recently, many open source libraries have implemented these schemes, such as HElib [13], PALISADE [14], SEAL [15] and HEAAN [16].

To alleviate the computation bottlenecks, some studies dedicated to exploring different acceleration strategies for polynomial arithmetic, such as adopting the Number Theoretic Transformation (NTT) and the Residue Number System (RNS) representation. SEAL employs the Harvey's algorithm [17] and follows the in-place iterative radix-2 implementation. The four-step Cooley-Tukey algorithm [18], [19] is another approach that was proposed to implement NTT, which is more friendly to memory because it allows splitting the N-point NTT into several ones with smaller size. The RNS is a commonly utilized technique to efficiently conduct computation over integers larger than the processor word-size. In practice, there are some RNS variant of these schemes [20], [21], [22] and most HE libraries implement these variants.

Previous works on accelerating FHE schemes mostly focus on high-performance platforms, by increasing the parallelism of computation. For example, Boemer et al. proposed Intel HEXL [23] that accelerated the polynomial arithmetic using the Intel Advanced Vector Extensions 512 (AVX512) instruction set and achieved a $7.2\times$ speedup, which is now integrated into SEAL and PALISADE. Other works concentrated on improving the computation concurrency through general-purpose GPU (GPGPU) implementation. The main idea is to first accelerate polynomial arithmetic, where several methods have been implemented on the GPU, including the generalized DGT-based algorithm [24], the Harvey's algorithm [25], the four-step Cooley-Tukey algorithm [26], [27], [28], and the base conversion [29]. Based on this, further acceleration of HE functions using methods like kernel coalescence [28] was obtained, such as $5\times$-$22\times$ for homomorphic operations in [24] and $257\times$ for CKKS bootstrapping in [28]. Meanwhile, prior works proposed many hardware accelerators[30], [31], such as the CraterLake[31], which achieved $4,600\times$ speedup compared to a 32-core CPU. FHE schemes were not introduced to the IoT systems, until Natarajan et al. proposed the first CKKS implementation in the embedded domain, i.e., SEAL-Embedded [32]. They concentrated on reducing the memory consumption of CKKS encoding and encryption while maintaining the performance and used an adapter as a middle layer to convert the ciphertexts sent by the devices into a form compatible to SEAL.

So far, although embedded GPUs have been deployed in many IoT devices, there is no optimized implementation of FHE schemes targeted for this platform. Previous works on accelerating FHE schemes using GPU only support a portion of the parameters and are not general enough to cover all applications. Additionally, many open-source libraries and works adopt non-constant-time implementations, such as SEAL [15] and [28]. A recent side-channel attack on SEAL v3.2 [33], although not affecting implementations after v3.3, raised awareness of the potential security risks that non-constant-time implementations may bring.

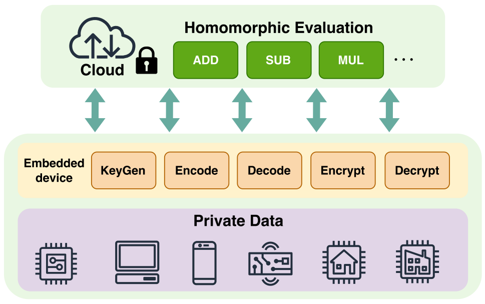


Fig. 1. Secure device-to-cloud computation model of CARM.

1.2 Contributions and Road-Map

In this work, we propose CARM, the first optimized GPU implementation of word-wise FHE schemes in IoT scenario based on SEAL v4.0 library [15] that covers BGV, BFV and CKKS. Our library provides fast homomorphic multiplication for two types of devices in the IoT network topology, i.e., middle-layer and cloud devices. Our solution has two advantages: first, we accelerate the performance of homomorphic multiplication on cloud devices, and second, we allow middle-layer devices to provide more real-time evaluation results, thus relieving the computational pressure on cloud devices. We follow the secure deployment solution for IoT [32], and the computation model is shown in Fig. 1. In summary, we make the following contributions:

- We provide constant-time low-level arithmetic implementation written in the CUDA parallel thread execution (PTX) instructions with minimum instructions and register usage, covering multiply-accumulation, conditional subtraction, Barrett reduction, and butterfly operation, which allow us to eliminate the divergence in warps and provide side-channel resistance.
- We exploit a parametric hierarchical design for the (I)NTT with a mix-type memory usage strategy to fully utilize the on-board GPU memory and reduce the IO latency. We offer the implementation of both performance-prior (PP) and memory-prior (MP) versions, allowing a flexible choice according to the target platforms.
- We devise a generic design of homomorphic multiplication and achieve a balance between the instruction- and thread-level parallelism through kernel coalescence (KC). For that KC will not always improve the performance, we illustrate its impact and give the bound between the methods with or without KC. We also exploit the KC approach in the BEHZ-type multiplication that reduces the memory access and meanwhile preserves good parallelism.
- We deploy our GPU implementation on two devices, targeting two different scenarios. Compared with the

CPU baseline, our (I)NTT implementation improves the performance up to $148.4\times$ for a single (I)NTT and $305.4\times$ for batched (I)NTT with size 21 on a Tesla V100S GPU, and up to $5.7\times$ on a Jetson AGX Xavier GPU. With the MP configuration, we show a $170.7\times$ improvement in the size of the precomputed table. We also achieve up to $378.4\times$, $233.6\times$, and $287.2\times$ on Tesla V100S ($8.8\times$, $9.2\times$, and $10.3\times$ on Jetson AGX Xavier) for the multiplication of BGV, BFV and CKKS, respectively.

The rest of this paper is organized as follows. In Section 2, we present some definitions and introductions to R-LWE-based FHE schemes, GPU, and the programming model. In Section 3, we describe the structure of CARM as well as the considerations and trade-offs in our design. The implementation and optimization details are presented in Section 4. Thereafter, we provide the performance results, comparison, and some further discussions in Section 5. Finally, we conclude this paper in Section 6.

2 Preliminaries

2.1 Notation

Let N be a power of 2. We choose the $2N$-th cyclotomic polynomial x^N+1 to define the quotient ring $R=\mathbb{Z}[x]/(x^N+1)$, whose elements are integer polynomials of degree less than N and denoted by lower-case boldface letters, e.g., $\boldsymbol{p}$ or $p(x)$. We use $\lfloor\cdot\rfloor$, $\lceil\cdot\rceil$, and $\lfloor\cdot\rceil$ to denote the rounding down, up, and to the nearest integer of a real number, respectively, and use $[\cdot]_q$ (for q an integer) to denote the centered remainder of an integer modulo q. When these operations are applied to a polynomial, we indicate that the corresponding operation is performed on each coefficient separately. By $a\overset{\$}{\leftarrow}S$, we denote that a is sampled uniformly from a finite set S. For a distribution $\mathcal{X}$ on S, we use $a\leftarrow\mathcal{X}$ to denote the sampling of a from S according to the distribution $\mathcal{X}$.

2.2 BGV, BFV and CKKS

The BGV, BFV and CKKS are three prominent word-wise FHE schemes. This type of scheme supports batched computation by pre-splitting the input into different slots and is efficient on some time-consuming operations. The BGV and BFV work on the same plaintext space that perform exact computations over finite fields, while the CKKS supports approximate evaluation of real numbers at a preset precision. The security of BGV, BFV and CKKS is based on the hardness of the (Ring-)Learning with Errors ((R-)LWE) problem. Let λ denote the security parameter. Let $\boldsymbol{s}$ be a random element in R_q and $\mathcal{X}=\mathcal{X}(\lambda)$ be a distribution over R_q. The decision R-LWE problem is defined as to distinguish the samples $(\boldsymbol{a}_i,\boldsymbol{b}_i)$ from the uniform distribution on R_q^2, where $\boldsymbol{b}_i=\boldsymbol{a}_i\boldsymbol{s}+\boldsymbol{e}_i$, $\boldsymbol{a}_i\overset{\$}{\leftarrow}R_q$, $\boldsymbol{e}_i\leftarrow\mathcal{X}$.

Let t be the plaintext modulus, $Q=\prod_{i=0}^{L}q_i$ be the ciphertext modulus and $Q_l=\prod_i^l q_i$ be the modulus for the ciphertext at level l, where q_i are primes. The special modulus [34] is denoted as p. We define the main components of the three HE schemes as follows.

- Key generation. Given the system parameter $\texttt{params}=(\lambda,N,Q)$ and distributions $\mathcal{X}_{\text{key}}$ and $\mathcal{X}_{\text{err}}$, generates the public key and secret key in the following way:
  - Secret key. Sample $\boldsymbol{s}\leftarrow\mathcal{X}_{\text{key}}$, and set the secret key $\texttt{sk}:=(1,\boldsymbol{s})$.
  - Public key. Sample $\boldsymbol{a}\overset{\$}{\leftarrow}R_Q$ and $\boldsymbol{e}\leftarrow\mathcal{X}_{\text{err}}$. The public key is formed as $\texttt{pk}:=([-\boldsymbol{a}\cdot\boldsymbol{s}+t\boldsymbol{e}]_Q,\boldsymbol{a})$ in BGV, and as $\texttt{pk}:=([-\boldsymbol{a}\cdot\boldsymbol{s}+\boldsymbol{e}]_Q,\boldsymbol{a})$ in both BFV and CKKS.
- Encryption. Given a public key $\texttt{pk}=(\boldsymbol{u}_0,\boldsymbol{u}_1)\in R_Q^2$ and a message $m\in R$, sample $\boldsymbol{r}\leftarrow\mathcal{X}_{\text{key}}$ and $\boldsymbol{e}_0,\boldsymbol{e}_1\leftarrow\mathcal{X}_{\text{err}}$. The resulting ciphertext of BGV, BFV, or CKKS encryption is described respectively as follows:

  $\texttt{BGV.Enc}(\texttt{pk},m)=([[m]_t+\boldsymbol{r}\cdot\boldsymbol{u}_0+t\boldsymbol{e}_0]_Q,[\boldsymbol{r}\cdot\boldsymbol{u}_1+t\boldsymbol{e}_1]_Q)$;

  $\texttt{BFV.Enc}(\texttt{pk},m)=([\Delta_{\texttt{BFV}}\cdot[m]_t+\boldsymbol{r}\cdot\boldsymbol{u}_0+\boldsymbol{e}_0]_Q,\ [\boldsymbol{r}\cdot\boldsymbol{u}_1+\boldsymbol{e}_1]_Q)$, where $\Delta_{\texttt{BFV}}=\lfloor Q/t\rfloor$;

  $\texttt{CKKS.Enc}(\texttt{pk},m)=([m+\boldsymbol{r}\cdot\boldsymbol{u}_0+\boldsymbol{e}_0]_Q,[\boldsymbol{r}\cdot\boldsymbol{u}_1+\boldsymbol{e}_1]_Q)$.
- Decryption. Given a secret key $\texttt{sk}=(1,\boldsymbol{s})$ and a ciphertext $\texttt{ct}=(\boldsymbol{c}_0,\boldsymbol{c}_1)\in R_{Q'}^2$, the decryption algorithm is like the following:

  $\texttt{BGV.Dec}(\texttt{sk},\texttt{ct})=[\boldsymbol{c}_0+\boldsymbol{c}_1\cdot\boldsymbol{s}]_{Q'}$;
  $\texttt{BFV.Dec}(\texttt{sk},\texttt{ct})=\lfloor t/Q'\cdot[\boldsymbol{c}_0+\boldsymbol{c}_1\cdot\boldsymbol{s}]_{Q'}\rceil$;
  $\texttt{CKKS.Dec}(\texttt{sk},\texttt{ct})=[\boldsymbol{c}_0+\boldsymbol{c}_1\cdot\boldsymbol{s}]_{Q'}$.

  Note that for leveled schemes, the modulus Q' is equal to Q_l at level l, while for the scale-invariant scheme BFV, the modulus will not change.
- Evaluation. Homomorphic encryption provides a way to perform operations on ciphertext without decryption. Common homomorphic evaluations include homomorphic addition and multiplication. For the three schemes, we give their detailed evaluation procedures below.
  - Addition. For any BGV, BFV, or CKKS schemes, given two ciphertexts $\texttt{ct}$ and $\texttt{ct}'$ in $R_{Q'}^2$, their sum is defined as the follows:

    $\texttt{Add}(\texttt{ct},\texttt{ct}')=[\texttt{ct}+\texttt{ct}']_{Q'}$.
  - Multiplication. Given two ciphertexts $\texttt{ct}=(\boldsymbol{c}_0,\boldsymbol{c}_1)$, $\texttt{ct}'=(\boldsymbol{c}_0',\boldsymbol{c}_1')$ in $R_{Q'}^2$, the product of which yields a triple defined as the follows:

    $\texttt{Mult}(\texttt{ct},\texttt{ct}')=(\tilde{\boldsymbol{c}}_0,\tilde{\boldsymbol{c}}_1,\tilde{\boldsymbol{c}}_2)$.

    Here $(\tilde{\boldsymbol{c}}_0,\tilde{\boldsymbol{c}}_1,\tilde{\boldsymbol{c}}_2)=[(\boldsymbol{c}_0\cdot\boldsymbol{c}_0',\boldsymbol{c}_0\cdot\boldsymbol{c}_1'+\boldsymbol{c}_1\cdot\boldsymbol{c}_0',\boldsymbol{c}_1\cdot\boldsymbol{c}_1')]_{Q'}$ for BGV and CKKS, and $[\lfloor t/Q'\cdot(\boldsymbol{c}_0\cdot\boldsymbol{c}_0',\boldsymbol{c}_0\cdot\boldsymbol{c}_1'+\boldsymbol{c}_1\cdot\boldsymbol{c}_0',\boldsymbol{c}_1\cdot\boldsymbol{c}_1')\rceil]_{Q'}$ for BFV.

2.3 Basic FHE Operations

The basic operations of RLWE-based fully homomorphic encryption schemes are performed on polynomials. We provide here the detailed procedures of some of these functions, which enable the efficiency of the schemes.

2.3.1 *Barrett Reduction*

To perform a 128-bit integer modular reduction with a 64-bit storage unit, we implement modified Barrett reduction [35]. Given a modulus q, a precomputed value $\gamma=\lfloor 2^{128}/q\rfloor$, and an integer $a<2^{128}$, a and γ are represented as $a=2^{64}\cdot a_1+a_0$ and $\gamma=2^{64}\cdot\gamma_1+\gamma_0$, respectively ($a_1,a_0,\gamma_1,\gamma_0$ are all 64-bit integers). Then, the reduction computes

$$a - \left\lfloor \frac{a_1\gamma_1 \cdot 2^{128} + (a_1\gamma_0 + a_0\gamma_1) \cdot 2^{64} + a_0\gamma_0}{2^{128}} \right\rfloor \cdot q,$$

which equals to

$$a - \left\lfloor \frac{(a_1 \cdot 2^{64} + a_0) \cdot (\gamma_1 \cdot 2^{64} + \gamma_0)}{2^{128}} \right\rfloor \cdot q = a - \left\lfloor \frac{a \cdot \gamma}{2^{128}} \right\rfloor \cdot q.$$

This result is actually either $[a]_q$ or $[a]_q + q$, and then we can get the final reduced result in a constant-time manner via a conditional subtraction by q.

2.3.2 Number Theoretic Transform

To include fast polynomial multiplication, Number Theoretic Transform (NTT) is utilized. In our setting, polynomials are multiplied modulo $x^N + 1$, and thus we perform a negacyclic version of NTT accordingly. To be specific, for a polynomial $\boldsymbol{f} \in R_q$ with $2N|(q-1)$, there exists a primitive $2N$-th root of unity $\omega \in \mathbb{Z}_q$, and the negacyclic NTT transform of $\boldsymbol{f} = (f_0, f_1, \cdots, f_{N-1})$ is defined as

$$\hat{\boldsymbol{f}} = (\hat{f}_0, \hat{f}_1, \cdots, \hat{f}_{N-1}),\ \hat{f}_i = \sum_{j=0}^{N-1} f_j \omega^{j(2i+1)}, i \in [0, N).$$

And the inverse, INTT, is performed by replacing the term ω here with ω^{-1} and we omit it for simplicity. Upon the NTT technique, the product of two polynomials $\boldsymbol{f}, \boldsymbol{g} \in R_q$ can be computed via $\texttt{INTT}(\texttt{NTT}(\boldsymbol{f}) \circ \texttt{NTT}(\boldsymbol{g}))$, where $\circ$ denotes element-wise multiplication of vector.

2.4 Residue Number System

The Residue Number System [36], [37] is a common representation used in implementing HE schemes [13], [14], [15], [16], where the modulus Q_l is set to be a product of several small pairwise coprime values $q_0, \cdots, q_l$. As we use NTT for fast polynomial multiplication, each q_i is required to be a prime number by this technique. This implies a ring isomorphism $R_{Q_l} \simeq R_{q_0} \otimes \cdots \otimes R_{q_l}$. Under this representation, we can convert the operation modulo Q_l, which is larger than the machine word-size, into l independent operations over smaller moduli $q_0, \cdots, q_l$, by executing it residue-wisely. This offers an efficient way to reduce the arithmetic size and improve the parallelism of HE operations, especially for modular addition and multiplication.

Some operations are hard to be compatible with RNS, such as the division and rounding in BFV. Two RNS-variants [20], [21] are proposed to solve it, where the fast base conversion is applied to switch the RNS base. It allows to convert residues of $x \in \mathbb{Z}_Q$ from base Q to P efficiently. In this paper, we employ the algorithm proposed in [20], which is formulated as

$$\texttt{FastBconv}_{Q\to P}(x) = \left(\left[\sum_{i=0}^{l} \left[x_i \frac{q_i}{Q} \right]_{q_i} \times \frac{Q}{q_i} \right]_{p_j} \right)_{j=0}^{v},$$

where $Q = \prod_0^l q_i, P = \prod_0^v p_i$ are two coprime moduli.

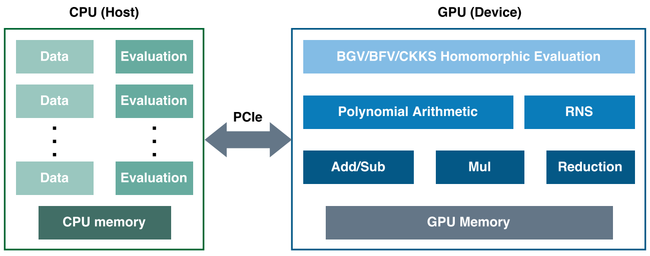


Fig. 2. The host-device model and library structure of CARM.

2.5 CUDA Programming Model

The GPU, a multi-core processor with highly parallel features, can execute the same program over multiple data in parallel through the interface Compute Unified Device Architecture (CUDA), which allows access to the GPU resource. In this programming mode, the CPU calls a kernel, which is executed in parallel by several CUDA threads.

CUDA follows a hierarchy with both execution units and memory. The minimum execution unit is a thread, and multiple threads (up to 1024) can be organized into 1- to 3-dimension, thus forming a block. Similarly, blocks can be arranged into 1- to 3-dimension grids. This hierarchy offers compatibility with the structure of operands, making the invocation of elements simpler. GPU contains three types of read-write memory, i.e., global memory (GMEM), shared memory (SMEM) and register file (RF), and two types of read-only memory, i.e., constant memory and texture memory. Threads within a block have access to the block SMEM, and each thread has private RF. The GMEM and read-only memory is accessible to all threads, while the latency is higher compared to the SMEM and RF. During the kernel execution, each block has to be executed independently in a sequence. Consecutive 32 threads in a block belong to the same warp, which is scheduled by the warp schedulers (WSs) in the streaming multiprocessors (SMs).

3 Implementation Overview

3.1 Library Structure

Our CARM library is proposed to accelerate HE schemes on both middle-layer and cloud devices, to provide efficient homomorphic evaluation of privacy data. The structure of CARM is shown in Fig. 2. Our design follows the host-device (HD) computation model, where the CPU launches a kernel and the GPU computes and returns the result. In functionality, CARM can be divided into two parts, one contains the initialization of the GPU memory and pre-computation of some constants according to the scheme parameters, such as the NTT table; the other contains the implementation of the functions of the FHE scheme, which consists of three layers: the low-level arithmetic layer, polynomial operations layer and the scheme layer.

The basic layer contains the modulo reduction, addition, and multiplication. We provide constant-time implementation written in CUDA PTX assembly and well optimize it by minimizing the instruction numbers and register usage. The middle layer consists of polynomial arithmetic and RNS tools to accelerate the computation of the scheme functions. At the top layer, CARM offers optimized homomorphic evaluation kernels for BGV, BFV and CKKS. Additionally, our library supports $|N| \in [11, 17]$, which covers the parameter requirements of most applications.

3.2 Design Considerations and Trade-Offs

In this section, we illustrate the considerations and trade-offs in the design of CARM. We concentrate on reducing time-consuming operations such as data access and synchronization and provide many trade-offs between the cost of computation and resource.

3.2.1 Memory Usage

The computation of FHE functions is in nature memory-bound since they feature low arithmetic intensity but require plenty of data access. In this work, we reduce both memory usage and IO latency. For the low-level arithmetic kernel, we make full use of the RF and SMEM and minimize the number of temporary registers. Based on this, we coalesce multiple kernels to large ones for high-level homomorphic operations to reduce data transfer, especially the GMEM access. For example, the (I)NTT and point-wise multiplication are coalesced into one kernel, which is illustrated in Section 4.4.1. Note that after coalescence, a function may still contain more than one kernel, such as in Section 4.4.2. This preserves some data transfer, but prevents resource overuse or overflow that lead to low occupancy and performance degradation.

3.2.2 Synchronization

Inserting a synchronization point is a commonly applicable method to prevent data conflict when data interaction exists. Throughout this work, we arrange the order of reading and writing data appropriately to eliminate some expensive synchronization. Particularly, considering that there is no need to synchronize the threads in the same warp because there is no data conflict, we eliminate the thread synchronization in five layers in (I)NTT through finely tune the inner loop.

3.2.3 ILP and TLP

For memory-bound computation, reusing data and having one thread execute more operations provide a way to increase the instruction-level parallelism (ILP) and reduce IO latency, thus improving the overall performance. Limited by the fixed on-board memory on GPU, an SM cannot execute multiple blocks with high memory overhead simultaneously, for this will cause some warp schedulers to remain in the idle state and decrease the thread-level parallelism (TLP). However, low SM occupancy does not always lead to low performance, as storing more data in the SMEM may reduce access to the GMEM, which may consume much more cycles than arithmetic operations. In this work, we choose suitable ILP and TLP for implementations of different parameters, which lead to peak kernel performance. For example, in the (I)NTT, we instantiate the two kernels with different ILP and form the kernels $\mathcal{K}_{l_1}$ and $\mathcal{K}_{l_2}$ as 8 and 2 per-thread implementation respectively. Thus, we can reduce the memory interaction in $\mathcal{K}_{l_1}$ and reserve more RF in $\mathcal{K}_{l_2}$ to store the intermediate element in the generic RNS multiplication kernel, which is described in Section 4.4.1.

4 Implementation Details and Optimizations

In this section, we present the implementation details of CARM, as well as our optimization techniques and trade-offs. First, we give the constant-time low-level arithmetic implementation written in the CUDA PTX assembly, based on which we implement the polynomial operations, including the (I)NTT and the RNS tools. Then, we optimize the homomorphic multiplication using techniques such as kernel coalescence. We strive for a generic, scalable and parametric design that is adaptable according to the platforms, and we provide several strategies for making choices.

4.1 Constant-Time Arithmetic

The efficient implementation of the low-level arithmetic ensures the performance, and keeping the execution time constant provides the ability to resist side-channel attacks. Here, we provide our optimized inline functions, including multiply-accumulation, conditional subtraction, and Barrett reduction, and show our methods to minimize the number of instructions and register usage.

4.1.1 Multiply-Accumulation

The multiplication of two 64-bit elements requires two PTX instructions on the GPU, so that the two 64-bit halves of the result are obtained. For the multiply-accumulate operation, unlike other methods that perform addition after multiplication, we use the `mad` and `madc` instructions to simplify the process, which is given in Algorithm 1. Specifically, we split the 128-bit addition and merge it with the 64-bit multiplication. The other issue that needs to be addressed is the carry. We store the carry flag implicitly in the low 64-bit multiply-accumulate operation, and then use the `madc` instruction to add the carry flag in the high 64-bit multiply-accumulate operation. In this way, we get the result without introducing additional overhead, as the number of instructions used is the same as the original multiplication.

Algorithm 1. `Multiply_Accumulate`

```
Input: $acc_1, acc_0, a, b \in [0, 2^{64})$
Output: $acc'_1, acc'_0 \in [0, 2^{64})$
1: mad.lo.cc.u64   $acc_0, a, b, acc_0$
                       $\triangleright\ acc'_0 = acc_0 + (a \cdot b)_{lo}$
2: madc.hi.u64   $acc_1, a, b, acc_1$
                       $\triangleright\ acc'_1 = acc_1 + (a \cdot b)_{hi} + carry$
```

4.1.2 Conditional Subtraction

SEAL uses a logic branch to control the condition in subtraction-based reduction. Namely, it reduces numbers from $[0, 2q)$ to $[0, q)$ by subtracting q from elements greater than q and remaining the others unchanged. However, this unbalanced execution poses a potential threat of side-channel attacks and introduces thread divergences in warps, which will incur a drop in the instruction throughput as the different execution paths must be serialized. The instructions with a guard predicate are commonly used for implicit comparisons. Nevertheless, the predicate variable prefixed to the next instruction still specifies a conditional execution. To mitigate the problems, we follow [38] to exploit the arithmetic and shift instructions and subtract q from the input, and the sign gives the implicit comparison result, which can be obtained by logic shifting. Through this approach, we can conduct a conditional subtraction in constant-time and eliminate the divergence in warps caused by the previous control flow instructions.

4.1.3 Barrett Reduction

The main idea of Barrett reduction is to transform division to fast arithmetics like multiplication and shifting. The multiplication of two 64-bit elements (i.e., $a = x \cdot y$) yields a 128-bit product, so we implement both 64-bit and 128-bit Barrett reduction using CUDA PTX assembly. In detail, CARM offers three types of Barrett reduction, one for reducing 64-bit numbers and two others for reducing 128-bit numbers, one of which is a faster variant proposed by Shoup [39] to reduce the computation overhead. The reduction of a 64-bit input a follows a conventional and commonly applicable way of computing $tmp_0 = \lfloor \frac{a \cdot \gamma_{64}}{2^{64}} \rfloor$ and subtract q multiple of it from a. However, when reducing the product of two 64-bit numbers, we can only compute and store the high 64-bit and low 64-bit of the 128-bit number and *ratio* separately, due to the limitation of CUDA computing capability. Here, we present a constant-time approach by using the combination of instruction `mul.hi.u64` and `mad.lo.cc.u64` to implicitly record the overflow flag in the condition code register and handle the carry bits of additions. For a pseudocode description of our implementation, see Algorithm 2. The faster version of the 128-bit reduction takes inputs of x and a precomputed number $\lfloor \frac{y \cdot 2^{64}}{q} \rfloor$, which trades the storage of $\lfloor \frac{y \cdot 2^{64}}{q} \rfloor$ for simplifying the computation of $\lfloor \frac{x \cdot y \cdot \gamma_{128}}{2^{128}} \rfloor$. As the output is in range $[0, 2q)$, we use the conditional subtraction approach illustrated before to ensure a constant-time execution. The ratio $\gamma_i = \lfloor \frac{2^i}{q} \rfloor$ is precomputed and stored in the GMEM. During the computation, we load the ratio into the RF and all the calculations are performed in the RF to achieve the minimum latency. Additionally, we reduce the use of temporary registers and reduce the number to one to minimize memory usage.

Algorithm 2. `Barrett_Reduction_128`

```
Input: $a = 2^{64} \cdot a_1 + a_0, \gamma = 2^{64} \cdot \gamma_1 + \gamma_0, q, 0 \le a, \gamma, q < 2^{64}$
Output: $res = a \bmod q, 0 \le res < 2q$
 1:  mul.hi.u64     $tmp, a_0, \gamma_0$
 2:  mad.lo.cc.u64  $tmp, a_0, \gamma_1, tmp$
                          $\triangleright\ c_0 = \lfloor ((a_0 r_0)_{hi} + (a_0 r_1)_{lo})/2^{64} \rfloor$
 3:  madc.hi.u64    $res, a_0, \gamma_1, 0$
 4:  mad.lo.cc.u64  $tmp, a_1, \gamma_0, tmp$
                          $\triangleright\ c_1 = \lfloor ((a_0 r_0)_{hi} + (a_0 r_1)_{lo} + (a_1 r_0)_{lo})/2^{64} \rfloor$
 5:  madc.hi.u64    $res, a_1, \gamma_0, res$
 6:  mad.lo.u64     $res, a_1, \gamma_1, res$
                          $\triangleright\ res = (a_0 r_1)_{hi} + (a_1 r_0)_{hi} + (a_1 r_1)_{lo}$
 7:  mul.lo.u64     $res, res, q$
 8:      sub.u64    $res, a_0, res$          $\triangleright\ res = a_0 - (res * q)_{lo}$
 9:      sub.s64    $res, res, q$
10:      shr.s64    $tmp, res, 63$
11:      and.b64    $tmp, tmp, q$
12:      add.s64    $res, res, tmp$
```

4.2 NTT Implementation and Trade-Offs

A full transformation of a $(N-1)$-degree polynomial takes $\log N$ layers, where N is a power of 2. Each layer contains

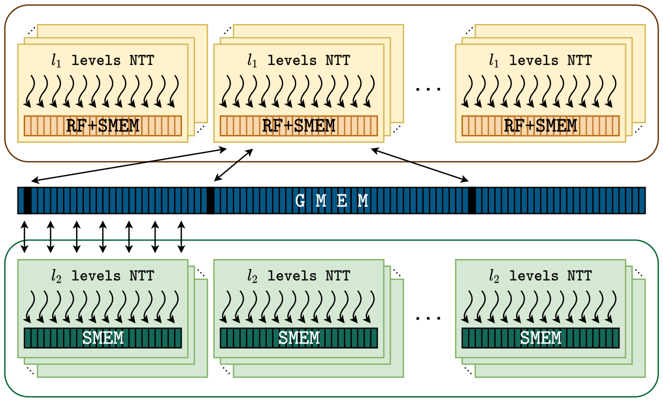


Fig. 3. Execution and memory access pattern of the proposed (I)NTT kernel.

the processing of $N/2$ butterfly operations, of which the inputs are two coefficients and the corresponding twiddle factor (TW). Throughout this paper, we apply the hierarchical structure [28] and divide the procedure into two steps, i.e., N_1-point NTT and N_2-point NTT, where $N = N_1 N_2$. This yields two kernels that comprise an (I)NTT execution and are responsible for processing different layers respectively, namely, l_1 and l_2 layers, where $\log N = l_1 + l_2$, $l_1 = \log N_1$ and $l_2 = \log N_2$. For compatibility, we provide a generic GPU implementation of (I)NTT that works for all cyclotomic rings with N in the range $\{2^{11}, 2^{12}, \cdots, 2^{17}\}$. Additionally, we implement two versions of (I)NTT, one focusing on better performance and the other on less resource usage, taking into account the features of different platforms in IoT scenarios.

Previous work shows that setting N_1 and N_2 close to $\sqrt{N}$ is the most efficient method [19], however, the IO latency is not taken into account. For example, in [28] the authors fixed $l_1 = 8$ and obtain 14- to 17-layer (I)NTT implementation by adjusting l_2. In each Kernel, they instantiate it with 256 threads and used a fixed 8 per-thread implementation, which requires the execution of a data interaction between RF and SMEM every three layers. We explore a mix-type memory usage strategy, which is different from the prior implementations [26], [28]. Fig. 3 shows the process and data access pattern of the kernels. Because INTT is the inverse of NTT, for the sake of brevity, we only describe the implementation of NTT in the following content.

4.2.1 Performance-Prior Version

Let λ denote the batch size of RNS-NTT and $\mathcal{K}_l$ denote the kernel that performs l layers NTT. We instantiate the $\mathcal{K}_{l_1}$ and $\mathcal{K}_{l_2}$ kernels with $\lambda N/\rho$ threads, where each thread performs ρ per-thread in-place NTT. The $\mathcal{K}_{l_1}$ kernel is responsible for executing 1 to 6 layers of the NTT. For those that require only three layers or less, we load the data directly from GMEM into RF and let $\rho = 2$, 4, and 8 corresponding to the cases where $l_1 \in \{1, 2, 3\}$, respectively. For $l_1 \in \{4, 5, 6\}$, we let $\rho = 8$ and exchange the data in the RF through the SMEM. In the $\mathcal{K}_{l_2}$ kernel, we set $l_2 = 11$, store the data in SMEM, and fix $\rho = 2$. With this instantiation, we can set aside more RF resources for other operations.

4.2.2 *Memory-Prior Version*

To perform an N-point NTT with a batch size β, it requires a $N\beta/128$ KB and $N\beta/64$ KB of storage, respectively, to hold the coefficients and precomputed table of `uint_64` type, e.g., 3β MB in total in the case $N = 2^{17}$. However, there may not be enough resources for storage-constrained devices, making the execution impossible. Several prior works [32], [40] proposed generating the twiddle factors in an on-the-fly manner to reduce the size of precomputed table. In detail, instead of generating all twiddle factors, this method only needs to store some of them in advance, and the rest can be generated by these. In [40], the authors applied a base-1024 approach on the last one or two layers, and the number of precomputed twiddle factors can be at most reduced to $(N/4 + (1024 + N/1024))\beta$, in which the computation of $x_O = \omega \times x_I$ is performed by calculating $x' = \omega_1 \times x_I$ and $x_O = \omega_2 \times x'$, where $\omega = \omega_1\omega_2$. However, this approach does not minimize the table size.

Algorithm 3. CT Butterfly in CUDA PTX Assembly

```
Input: $0 \leqslant X, Y < 4q$, $\varpi = 2^{64}$, $q < \varpi/4$, $0 < W < q$, $W' = \lfloor W\varpi/q \rfloor$
Output: $0 \leqslant X', Y' < 4q$
 1:  sub.s64     $X, X, 2q$
 2:  shr.s64     $T, X, 63$
 3:  and.b64     $T, T, 2q$
 4:  add.s64     $X, X, T$        ▷ $(X > 2q)\ ?\ X - 2q : X$
 5:  mul.hi.u64  $T, Y, W'$
 6:  mul.lo.u64  $T, T, q$
 7:  mul.lo.u64  $V, Y, W$
 8:  sub.u64     $V, V, T$        ▷ $V = [WY]_\varpi - [\lfloor \frac{W'Y}{\varpi} \rfloor \cdot q]_\varpi$
 9:  add.u64     $Y, X, 2q$
10:  sub.u64     $Y, Y, V$        ▷ $Y' = X - V + 2q$
11:  add.u64     $X, X, V$        ▷ $X' = X + V$
```

In our implementation of the memory-prior version, we follow [40] while exploiting a base-$\mathcal{D}$ approach, where $\mathcal{D}$ is a power of 2. We adjust the value of $\mathcal{D}$ to $\log \mathcal{D} = \frac{1}{2}\log N$ so that the number of precomputed twiddle factors is optimal for every N, i.e., $\mathcal{D} + N/\mathcal{D}$. Taking $N = 2^{17}$ as an example, in which case $\mathcal{D} = 256$, through this method, we can reduce the precomputed table size from 2β MB to 12β KB, which means a $170.7\times$ improvement.

4.2.3 *Constant-Time PTX Butterfly*

For the butterfly operation, we apply the Harvey's algorithm [17] and build inline device functions with CUDA PTX instructions. To eliminate the modular multiplication of n^{-1} in the INTT, we follow [25], [28] and modify the precomputed table of INTT by multiplying with $\frac{1}{2}$. In GS butterfly this modification only affects one operand, thus prior works [25], [28] performed division by 2 on the other operand with a 1-bit shift to the right, where odd numbers need to be added with q to ensure correctness. However, it is risky because the parity of the current operand can be inferred through energy analysis, thereby leaking information. To solve this, we multiply the least significant bit of the operand by q and add the result to it. This introduces one multiplication, but ensures constant-time execution.

Algorithm 4. GS Butterfly in CUDA PTX Assembly

```
Input: $0 \leqslant X, Y < 2q$, $\varpi = 2^{64}$, $q < \varpi/4$, $0 < W < q$, $W' = \lfloor W\varpi/q \rfloor$
Output: $0 \leqslant X', Y' < 2q$
 1:  add.u64     $T, X, 2q$
 2:  sub.u64     $T, T, Y$
 3:  add.u64     $X, X, Y$
 4:  sub.s64     $X, X, 2q$
 5:  shr.s64     $V, X, 63$
 6:  and.b64     $V, V, 2q$
 7:  add.s64     $X, X, V$
 8:  and.b64     $V, T, 1$        ▷ Judge parity
 9:  mul.lo.u64  $V, V, q$
10:  add.u64     $X, X, V$
11:  shr.s64     $X, X, 1$        ▷ $X' = \frac{(X+Y)}{2}$
12:  mul.hi.u64  $V, T, W'$
13:  mul.lo.u64  $V, V, q$
14:  mul.lo.u64  $Y, T, W$
15:  sub.u64     $Y, Y, V$        ▷ $Y' = \frac{W(X-Y)}{2}$
```

4.3 Fast Base Conversion

The fast base conversion `FastBconv` [20] offers an efficient approach to convert the base of the residues, of which the new base is coprime to the original base Q. In the SEAL library, the polynomials are stored in RNS representation, which forms a matrix of size $|Q| \times N$, and the addresses of the residues of the same modulus are contiguous. Let $\mathbf{I}$ denote the input matrix of residues base Q, $\mathbf{O}$ denote the output matrix of residues under base P, and $\mathbf{C}$ denote the conversion matrix. The SEAL library conducts an implicit matrix transposition through the iterators and obtains the conversion results through computing $\mathbf{I}^T \cdot \mathbf{C} = \mathbf{O}^T$. However, this transposition triggers the problem of non-unit-stride global memory access on the GPU, which should be avoided because of its high IO latency.

For this consideration, we form a 3-dimension kernel with N threads and change the computation to $\mathbf{C}^T \cdot \mathbf{I} = \mathbf{O}$, where the x-, y-, and z-axis are set based on the block size, $|Q|$, and ciphertext size, respectively. Each thread takes a row of the $\mathbf{C}^T$ matrix and a column of the $\mathbf{I}$ matrix, performs the inner product by an internal loop, and stores the result in the RF. We use two 64-bit registers to form a 128-bit accumulator to store the result.

A special case is that the BEHZ variant of BFV [20] requires converting the residues from base Q_l to base $\mathcal{B}_{\text{sk}}$ with an extra modulus $\tilde{m}$ (see Algorithm 5). In SEAL, it is performed by calling two individual conversions. We combine these two processes in our GPU implementation, i.e., the i-th thread takes the residues modulo p_i and $\tilde{m}$, and one column of the $\mathbf{I}$ matrix, performs two vector inner products and stores the results in two 128-bit accumulators, respectively. This approach reduces the memory access but increases the computational overhead of the threads, since the product of the residues of modulo $\tilde{m}$ by the $\mathbf{I}$ matrix is repeated by all threads, but due to the parallelism of the computation, it introduces no additional time consumption.

4.4 Homomorphic Multiplication

In the SEAL library, polynomials are represented in different forms, i.e., only CKKS keeps the ciphertext elements in

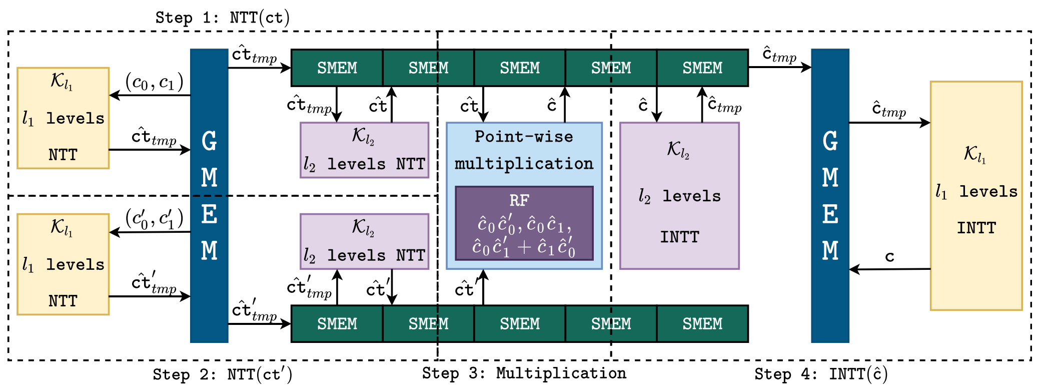


Fig. 4. Coalesced homomorphic multiplication kernel.

the NTT form. To reduce the computation overhead, the (I) NTT should be conducted for the ciphertexts of BGV and BFV. Meanwhile, the design of BFV makes it require more operations in multiplication, such as base conversion, which is different from BGV. We focus on the multiplication of the three scheme, and the following relinearization and mod-switching procedure are not included.

We define the homomorphic multiplication of two ciphertexts as $\mathtt{c} := \mathtt{Mult}(\mathtt{ct}, \mathtt{ct}') = (\tilde{c}_0, \tilde{c}_1, \tilde{c}_2)$, where $\mathtt{c}, \mathtt{ct} = (c_0, c_1), \mathtt{ct}' = (c'_0, c'_1) \in R^2_{Q'}$. Here $(\tilde{c}_0, \tilde{c}_1, \tilde{c}_2) = [(c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1)]_{Q'}$ for BGV and CKKS, and $[\lfloor t/Q' \cdot (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1)\rceil]_{Q'}$ for BFV. For compatibility, we exploit a generic RNS multiplication (GRM) module that works for both BGV and BFV, which coalesces the NTT, element-wise product and INTT kernels into one kernel so that the interaction with GMEM can be reduced to the load and store operations required by the (I)NTT themselves. The logic flow and data access pattern is shown in Fig. 4.

4.4.1 Generic RNS Multiplication Module

Let $\hat{\mathtt{ct}} = \mathtt{NTT}(\mathtt{ct})$ and the variables containing subscript tmp be the intermediate value in the (I)NTT processing. In our GRM module, first, we load the ciphertexts $\mathtt{ct} = (c_0, c_1)$ and $\mathtt{ct}' = (c'_0, c'_1)$ from GMEM and transform them into NTT domain using the $\mathcal{K}_{l_1}$ and $\mathcal{K}_{l_2}$ kernel defined before. Then, each thread loads data to the RF, computes the triple $\hat{\mathtt{c}} = ([c_0 \cdot c'_0]_{Q_l}, [c_0 \cdot c'_1 + c_1 \cdot c'_0]_{Q_l}, [c_1 \cdot c'_1]_{Q_l})$ and stores the results back to the SMEM. After that, we perform INTT to the elements in the triple respectively. During the entire process, the remaining load and store operations over GMEM are that required in performing (I)NTT on ct, ct′ and c, which in nature cannot be omitted. To fully utilize the RF, we instantiate the $\mathcal{K}_{l_2}$ kernel with $l_2 = 11$ that forms the 2 per-thread in-place (I)NTT. Through this configuration we can reserve enough registers for the computation of the triple, which ensures no register overflow will occur that impact the correctness in launching kernel.

One thing to note is that, for implementations with different parameters, especially N, kernel coalescence does not always improve the performance. In the original procedure, i.e., multiplication is composed by several kernels containing the NTT, element-wise product and INTT, where blocks are scheduled by different SMs and executed in parallel. However, in the GRM module, they are executed sequentially. Therefore, for high performance platforms with a large number of SMs and a small N, it is more efficient not to use kernel coalescence. In order to obtain the bounds between the two methods, we have performed adequate experiments and given them in detail in Section 5.3.

Algorithm 5. RNS Homomorphic Multiplication in BFV [20]

```
Input: ct, ct′ in base $Q_l$
Output: c in base $Q_l$
1: $\widetilde{\mathtt{ct}} \leftarrow \mathtt{FastBconv}_{Q_l \to \mathcal{B}_{\mathrm{sk}} \cup \{\tilde{m}\}}(\mathtt{ct})$
2: $\widetilde{\mathtt{ct}}' \leftarrow \mathtt{FastBconv}_{Q_l \to \mathcal{B}_{\mathrm{sk}} \cup \{\tilde{m}\}}(\mathtt{ct}')$    ▷ Convert fast to base $\mathcal{B}_{\mathrm{sk}} \cup \{\tilde{m}\}$
3: $\overline{\mathtt{ct}} \leftarrow \mathtt{SmMRq}_{\tilde{m}}(\widetilde{\mathtt{ct}})$
4: $\overline{\mathtt{ct}}' \leftarrow \mathtt{SmMRq}_{\tilde{m}}(\widetilde{\mathtt{ct}}')$    ▷ Reduce $Q_l$-overflows
5: $\bar{\mathtt{c}} = \overline{\mathtt{ct}} \cdot \overline{\mathtt{ct}}'$    ▷ Compute in $Q_l \cup \mathcal{B}_{\mathrm{sk}}$ using GRM
6: $\bar{\mathtt{c}}' = \lfloor \frac{t}{Q_l} \cdot \bar{\mathtt{c}} \rceil$    ▷ Division and rounding
7: $\mathtt{c} \leftarrow \mathtt{FastBconvSK}_{\mathcal{B}_{\mathrm{sk}} \to Q_l}(\bar{\mathtt{c}}')$    ▷ Convert exactly to $Q_l$
```

4.4.2 Kernel Coalescence in BFV

Our generic design is also applicable to the full RNS homomorphic multiplication applied in the BEHZ variant of BFV [20], which is implemented in SEAL. Here, we follow the notation in [20], and denote the extended base as $\mathcal{B} \cup \{m_{\mathrm{sk}}\} \cup \{\tilde{m}\}$, or $\mathcal{B}_{\mathrm{sk}} \cup \{\tilde{m}\}$, where $\mathcal{B}_{\mathrm{sk}} = \mathcal{B} \cup \{m_{\mathrm{sk}}\}$, $\mathcal{B}$ is the auxiliary base with v moduli, and m_{sk} and $\tilde{m}$ are two extra moduli. As described in Algorithm 5 [20], this process consists of the fast base conversion `FastBconv`, Small Montgomery Reduction `SmMRq`, polynomial multiplication, and exact conversion `FastBconvSK` (the details can be found in [20]). Considering the operand structure and memory usage, we exploit the kernel coalescence approach to perform the entire process:

- `FastBconv` with `SmMRq`. We use the method described in Section 4.3. The coalesced kernel performs the conversion of ct and ct′ from base Q_l to $\mathcal{B}_{\mathrm{sk}} \cup \{\tilde{m}\}$, yielding new ciphertexts with Q_l-overflows, which are then reduced through `SmMRq` [20].
- Multiplication. This is processed through our GRM by setting the moduli to $Q_l \cup \mathcal{B}_{\mathrm{sk}}$.
- Division, rounding and `FastBconvSK`. This kernel first performs multiplication by t, division by Q_l and rounding in $\mathcal{B}_{\mathrm{sk}}$ to achieve approximate rounding, and then performs the Shenoy and Kumaresan like conversion [41] to convert exactly from $\mathcal{B}_{\mathrm{sk}}$ to the original base Q_l.

TABLE 1
Testing Environment

Target platform	Middle-layer device	Cloud device
GPU	Jetson AGX Xavier	Tesla V100S PCIe
GPU cores	512	5120
GPU frequency	1.37 GHz	1.60 GHz
SMs	8	80
GPU memory	32 GB (unified)	32 GB
Compute capability	7.2	7.0
CUDA version	11.4	11.6
CPU	NVIDIA Carmel 64-bit ARMv8.2	Intel Xeon Silver 4210R
CPU cores	8	10
CPU frequency	2.26 GHz	2.40 GHz
CPU memory	32 GB (unified)	32 GB
CPU Cache	8 MB L2 4 MB L3	10 MB L2 13.75 MB L3

In these kernels, a block consists of 128 threads and loads 128 columns of the input residue matrix, which is a multiple of 32, so that all memory write requests of a warp fall into distinct memory banks and will not cause bank conflict. Meanwhile, after coalescence the function consists of more than one kernel. This will reserve enough memory resources for other operations, thus reducing the overall IO latency.

5 Results and Comparison

5.1 Testing Environment

To evaluate the performance of CARM, we implement it on two NVIDIA GPUs, one is the Jetson AGX Xavier developer kit and the other is the Tesla V100S PCIe, targeting the embedded and server-grade platforms. The detailed testing environment hardware configurations and the corresponding platforms are summarized in Table 1. Our GPU implementation is based on Microsoft SEAL library v4.0 [15] and all implementations are profiled with the same build and execute environment. For all experiments, we compile the C ++ implementation using g++ 11.1 and the GPU implementation using CUDA 11 on Ubuntu 20.04.

All the parameters we use all reach 128-bit security according to the LWE estimator [42], except benching the batched (I)NTT with $|N| = 14$ and batch size $\beta = 21$, because we wanted to be consistent with the parameters selected in [40] for better comparisons of performance.

5.2 NTT

Our generic and scalable design of (I)NTT kernels provides several ways of combining $\mathcal{K}_{l_1}$ and $\mathcal{K}_{l_2}$ for different parameters. The $l_2 = 11$ with $\rho = 2$ configuration targets for less SMEM usage with a high SM occupancy, and reserve sufficient RF to store the temp element after coalescing multiple kernels into one. Tables 2 and 3 present the performance of our implementation of (I)NTT with the performance-prior (PP) configuration on Tesla V100S, which are different in the batch size, and the comparisons with the CPU baseline implemented in SEAL v4.0 [15] and related works [25], [28], [40]. Additionally, we provide the benchmark of both performance-prior (PP) and memory-prior (MP) configuration of (I)NTT on Jetson AGX Xavier, as well as the CPU baseline in Table 4. Since there is no (I)NTT implementation on embedded GPUs yet, we only list the comparison with the CPU implementation. Here, [25] is a recent published work that reports the speed of single (I) NTT with $|N| \in [11, 15]$ on Tesla V100, and in [40] the authors give the execution time of NTT with $|N| \in [14, 17]$ batch size $\beta = 21$ on NVIDIA Titan V, in which the twiddle factors of the last layer are computed on-the-fly. Because their code is closed-source, we use the data provided by the corresponding paper. The (I)NTT implementation in [28] fixes $l_1 = 8$ and adjusts l_2 to get 14- to 17-layer (I)NTT, where the SMEM is utilize for data exchange every 3 layers. It is generic and suitable for most cases. As the code is open-source, we run it in our platform for comparison. Note that the specifications of the two GPUs are almost same as V100S, except for the memory bandwidth. In our work and previous works, the time of data transfer between CPU and GPU is not counted.

Compared to the CPU baseline, our GPU implementation provides $2.0\times$ to $305.4\times$ speedup on V100S and $1.1\times$ to $5.7\times$ speedup on AGX Xavier. Meanwhile, the speedup factor grows rapidly with the polynomial degree and batch size,

TABLE 2
Performance of (I)NTT Implementation on Tesla V100S, CPU Baseline, [25], and [28] (Measured in μs)

$\|N\|$	Batch	Our work (PP)		CPU		[25]	[28]		Speedup
	size	NTT	INTT	NTT	INTT	NTT	NTT	INTT	versus CPU
11	1	11.3	12.3	35.1	24.8	12.5			$3.1\times/2.0\times$
12	1	15.3	15.4	71.8	54.0	22.5	-	-	$4.7\times/3.5\times$
13	1	15.4	16.4	151.2	115.0	27.0			$9.8\times/7.0\times$
14	1	17.4	18.4	294.9	242.8	29.0	18.8	20.2	$16.9\times/13.2\times$
15	1	16.4	16.4	633.6	486.1	39.0	21.2	22.2	$38.6\times/29.6\times$
16	1	16.4	17.4	1299.5	1022.7	-	21.9	23.4	$79.2\times/58.8\times$
17	1	18.5	19.5	2746.2	2176.0		24.4	25.8	$148.4\times/111.6\times$

TABLE 3
Performance of Batched (I)NTT Implementation on Tesla V100S, CPU Baseline, [40], and [28] (Measured in μs)

$\|N\|$	Batch	Our work (PP)		CPU		[40]	[28]		Speedup
	size	NTT	INTT	NTT	INTT	NTT	NTT	INTT	versus CPU
12	2	15.4	15.4	142.8	108.0				$9.3\times/7.0\times$
13	4	15.4	16.4	563.0	431.0	-	-	-	$36.6\times/26.3\times$
14	21	42.0	49.2	6264.0	4814.3	44.1	39.8	44.2	$141.9\times/97.9\times$
15	21	57.3	71.6	13137.5	10258.9	84.2	66.2	73.5	$229.3\times/143.3\times$
16	21	105.5	132.1	27682.2	21805.0	156.3	114.1	133.4	$262.4\times/165.1\times$
17	21	190.5	239.6	58182.5	46879.1	304.2	210.5	242.4	$305.4\times/195.7\times$

TABLE 4
Performance of Batched (I)NTT Implementation on Jetson AGX Xavier and CPU Baseline (Measured in μs)

$\|N\|$	Batch size	Our work (PP) NTT	Our work (PP) INTT	Our work (MP) NTT	Our work (MP) INTT	CPU NTT	CPU INTT	Speedup versus CPU
11	1	111.9	110.8	150.8	162.0	199.6	122.7	1.8×/1.1×
12	1	153.2	154.8	186.2	210.0	211.0	234.5	1.4×/1.5×
13	1	159.0	170.9	217.2	246.4	372.2	497.8	2.3×/2.9×
14	1	215.8	248.0	282.3	317.3	738.8	856.7	3.4×/3.5×
15	1	308.7	336.2	404.1	459.5	1384.3	1408.4	4.5×/4.2×
16	1	564.9	624.1	769.3	902.7	3011.6	3000.4	5.3×/4.8×
17	1	1079.4	1170.4	1559.8	1801.2	5931.1	6253.2	5.5×/5.3×
12	2	160.2	163.7	196.1	222.7	275.1	272.1	1.7×/1.7×
13	4	375.0	431.3	451.4	519.4	1284.4	1278.5	3.4×/3.0×
14	8	1440.5	1603.9	1801.5	2040.7	6520.8	5512.9	4.5×/3.4×
15	16	3944.3	4070.1	5562.1	6305.6	21108.4	23246.5	5.4×/5.7×

and we improve the performance up to $148.4\times$ and $305.4\times$ correspondingly for single and batched NTT when $|N| = 17$. Compared to [25], [28], [40], our implementation shows better performance under most of the parameter settings and offers a larger parameter set. Thus, it is in nature generic and scalable, and suitable for many scenarios.

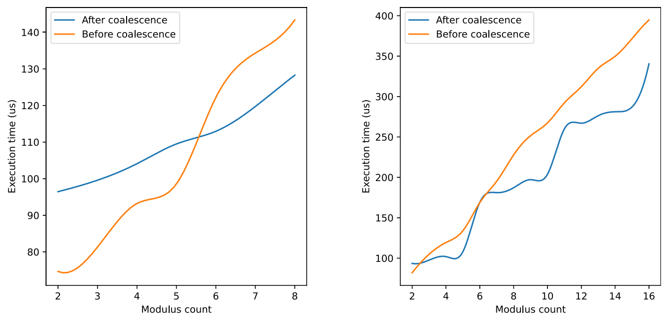
To measure the impact of our MP approach on the size of the precomputed TW tables, we present the results of different implementations in Table 5. Note that this is the size of the table needed for a single NTT, and the results of [40] are given as the smallest size that their implementation can lead to. Our implementation has a significant effect on reducing the table size, by a factor of 21.3 to 170.7 compared to the method without MP. Additionally, we achieve $20.1\times$ to $44.2\times$ improvements against [40].

5.3 Homomorphic Multiplication

In this section, we analyze the performance impact of our implementations using different optimization strategies. Because we believe that kernel coalescence does not improve the performance in all cases, we test the execution time of the two methods to obtain the bound, i.e., with or without GRM, respectively, where the results of $|N| = 14$ and 15 on the Tesla V100S are shown in Figs. 5a and 5b. From our experiments, we find that the method without GRM is faster when $|N|$ is less than 14. This result starts to change at $|N| = 14$ and the method with GRM outperforms the other when the number of moduli is greater than 6.

TABLE 5
The TW Table Size of the Baseline Implementation, [40], and Our MP Method (Measured in KB)

$\|N\|$	Baseline	[40]	Our work (MP)	Improvement versus CPU	Improvement versus [40]
11	32		1.5	21.3×	
12	64	-	2	32.0×	-
13	128		3	42.7×	
14	256	80.25	4	64.0×	20.1×
15	512	144.5	6	85.3×	24.1×
16	1024	273	8	128.0×	34.1×
17	2048	530	12	170.7×	44.2×



(a) $|N| = 14$ (b) $|N| = 15$

Fig. 5. Performance of the generic multiplication before and after coalescence.

Additionally, GRM shows more advantages when $|N|$ and the modulus count becomes larger. Throughout this work, we apply this bound and combine both methods to get better performance.

We evaluate the performance of our optimized homomorphic multiplication, where $|N|$ is the bit length of N and ℓ is the number of q_i, i.e., $\ell = l + 1$. Table 6 summarizes the execution time on a single-thread CPU, our GPU implementation on Tesla V100S, and some related works, where [24] and [28] are the state-of-the-art implementations of BFV and CKKS, and [43] and [44] are some recent works. The target platforms in these works have computing capability similar to ours, while our multiplication module does not contain the relinearization. Additionally, we provide the performance on Jetson AGX Xavier in Table 7. Compared to the CPU baseline, our GPU implementation provides up to $378.4\times$ speedup for BGV, $234.5\times$ speedup for BFV, and $287.2\times$ speedup for CKKS on Tesla V100S, and $8.8\times$, $9.2\times$, and $10.3\times$ on Jetson AGX Xavier.

5.4 Discussions

5.4.1 Resistance to Side-Channel Leakage

The butterfly and reduction operations play important roles in the implementation of HE schemes. Unfortunately, the majority open-source libraries and previous works use compiler-level optimization as a compensation to eliminate time variance. This poses a potential threat, as the study has shown that simple modifications to the code under this protection can reduce the hardness of the R-LWE problem [45]. In this paper, we use a constant-time design for all low-level arithmetic operations with the least possible performance loss. This technique brings an additional benefit, as an unbalanced execution will introduce thread divergences in warps, making different execution paths have to be serialized. We believe that such a countermeasure is worthwhile in two ways: first, it reduces the potential leakage of secret information; second, it increases the instruction throughput, which contributes to the efficiency.

5.4.2 Applicability of Proposed Approaches

Our design is generic and provides an optimization approach for GPU implementations of other homomorphic schemes and on other platforms. The difference of GPUs is mainly in terms of resource and computational capability. For different fully homomorphic encryption schemes, especially those based on

TABLE 6
Performance of Homomorphic Multiplication on Tesla V100S and Comparisons With Other Works (Measured in ms)

$\|N\|$	ℓ	BGV			BFV			CKKS			BFV[24]	BFV[43]	BFV[29]	CKKS[44]	CKKS[28]
		GPU	CPU	Speedup	GPU	CPU	Speedup	GPU	CPU	Speedup	Tesla P100	Tesla K80	Tesla V100	Tesla V100	Tesla V100
12	2	0.083	1.754	21.1×	0.167	4.049	24.2×	0.024	0.144	6.0×	1.833	1.214	0.997	-	
13	4	0.090	5.130	57.0×	0.189	16.213	85.8×	0.023	0.639	27.8×	3.538	3.061	1.178	0.403	
14	8	0.127	20.625	162.4×	0.386	71.727	185.8×	0.031	2.487	80.2×	11.747	13.914	2.412	0.742	-
15	16	0.355	86.342	243.2×	1.608	365.014	227.0×	0.059	9.988	169.3×			5.705	2.388	
16	32	1.080	361.504	334.7×	9.209	2159.234	234.5×	0.157	40.362	257.1×	-	-	59.473	33.577	17.4
17	32	1.988	752.253	378.4×	19.463	4546.390	233.6×	0.281	80.710	287.2×			-	-	7.96

TABLE 7
Performance of Homomorphic Multiplication on Jetson AGX Xavier (Measured in ms)

$\|N\|$	ℓ	BGV			BFV			CKKS		
		GPU	CPU	Speedup	GPU	CPU	Speedup	GPU	CPU	Speedup
12	2	1.048	4.363	4.2×	2.880	16.176	5.6×	0.173	0.723	4.2×
13	4	2.502	14.115	5.6×	8.800	65.036	7.4×	0.395	2.186	5.5×
14	8	9.549	56.164	5.9×	37.756	258.971	6.9×	1.006	8.862	8.8×
15	16	24.946	218.688	8.8×	112.884	1043.381	9.2×	3.380	34.726	10.3×

R-LWE, the low-level operations have low arithmetic density and share similar constructions. In our implementation, firstly, we propose some general designs with adjustable parameters that are scalable to be chosen according to the specific situation. Second, we provide both performance-prior and memory-prior approaches that can be applied to devices with different computational capabilities. Additionally, we optimize memory usage based on coalescing multiple kernels that reduces memory accesses. We also provide the impact of kernel coalescence on performance, giving a choice between different implementation approaches. In summary, our approach is meaningful for the optimization of schemes on various platforms.

6 Conclusion

In this work, we present CARM, the first optimized GPU implementation of word-wise homomorphic encryption schemes in IoT scenario, covering BGV, BFV and CKKS. Our work focuses on homomorphic multiplication, and offers various trade-offs between computational efficiency and memory consumption for deployment on different platforms. We provide evaluations on two different GPUs, targeting the embedded and cloud devices, and the results demonstrate the effectiveness of our approaches. Our generic and parametric design is applicable to GPU implementations of other functions, schemes, and on other platforms, which will be our future work.

References

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

[2] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 2009.

[3] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Des., Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, 2014.

[4] 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.

[5] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 2010, pp. 24–43.

[6] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," in *Proc. IEEE 52nd Annu. Symp. Found. Comput. Sci.*, 2011, pp. 97–106.

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

[8] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. 44th Symp. Theory Comput. Conf.*, 2012, pp. 1219–1234.

[9] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Proc. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2015, pp. 617–640.

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

[11] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. Innov. Theor. Comput. Sci. Conf.*, 2012, pp. 309–325.

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

[13] Helib, Apr. 2022. [Online]. Available: https://github.com/homenc/HElib

[14] Palisade, Apr. 2022. [Online]. Available: https://gitlab.com/palisade

[15] Microsoft SEAL (release 4.0), Apr. 2022. [Online]. Available: https://github.com/Microsoft/SEAL

[16] Heaan, Apr. 2022. [Online]. Available: https://github.com/snucrypto/HEAAN

[17] D. Harvey, "Faster arithmetic for number-theoretic transforms," *J. Symbolic Computation*, vol. 60, pp. 113–119, 2014.

[18] 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.

[19] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proc. ACM/IEEE Conf. Supercomput.*, 1989, pp. 234–242.

[20] J. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Proc. Sel. Areas Cryptogr.*, 2017, pp. 423–442.

[21] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Proc. Conf. Top. Cryptol.*, 2019, pp. 83–105.

[22] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Proc. Conf. Sel. Areas Cryptogr.*, 2019, pp. 347–368.

[23] F. Boemer, S. Kim, G. Seifu, F. D. M. D. Souza, and V. Gopal, "Intel HEXL: Accelerating homomorphic encryption with intel AVX512-IFMA52," in *Proc. 9th Workshop Encrypted Comput. Appl. Homomorphic Cryptogr.*, 2021, pp. 57–62.

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

[25] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," *J. Supercomput.*, vol. 78, pp. 2840–2872, 2021.

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

[27] A. A. Badawi, B. Veeravalli, and K. M. Mi Aung, "Faster number theoretic transform on graphics processors for ring learning with errors based cryptography," in *Proc. IEEE Int. Conf. Serv. Operations Logistics Inform.*, 2018, pp. 26–31.

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

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

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

[31] N. Samardzic et al., "CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 173–187.

[32] D. Natarajan and W. Dai, "Seal-embedded: A homomorphic encryption library for the Internet of Things," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2021, pp. 756–779, 2021.

[33] F. Aydin, E. Karabulut, S. Potluri, E. Alkim, and A. Aysu, "Reveal: Single-trace side-channel leakage of the seal homomorphic encryption library," *IACR Cryptol. ePrint Arch.*, vol. 2022, 2022, Art. no. 204.

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

[35] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Adv. Cryptol. Conf.*, 1986, pp. 311–323.

[36] H. L. Garner, "The residue number system," *IRE Trans. Electron. Comput.*, vol. 8, no. 2, pp. 140–147, 1959.

[37] A. Svoboda and M. Valach, "Operátorové obvody (operator circuits in czech)," *Stroje na Zpracovánı Informacı Inf. Process. Mach.*, vol. 3, pp. 247–296, 1955.

[38] J. W. Bos et al., "Crystals - kyber: A CCA-secure module-lattice-based kem," in *Proc. IEEE Eur. Symp. Secur. Privacy, EuroS&P*, 2018, pp. 353–367.

[39] V. Shoup, "NTL: A library for doing number theory," Tech. Rep., 2001. [Online]. Available: http:// www.shoup.net/ntl/

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

[41] A. P. Shenoy and R. Kumaresan, "Fast base extension using a redundant modulus in RNS," *IEEE Trans. Comput.*, vol. 38, no. 2, pp. 292–297, Feb. 1989.

[42] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, 2015. [Online]. Available: http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015–0016/jmc-2015–0016.xml

[43] P. G. M. R. Alves, J. N. Ortiz, and D. F. Aranha, "Faster homomorphic encryption over GPGPUs via hierarchical DGT," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2021, pp. 520–540.

[44] A. A. 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.

[45] N. Drucker and T. Pelleg, "Timing leakage analysis of non-constant-time NTT implementations with Harvey butterflies," *IACR Cryptol. ePrint Arch.*, vol. 2022, 2022, Art. no. 94.

Shiyu Shen is currently working toward the PhD degree with the School of Computer Science, Fudan university, Shanghai, China. Her research interests include lattice-based cryptography, post-quantum cryptography, homomorphic encryption, hardware design, and parallel processing.

Hao Yang is currently working toward the PhD degree with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. His research interests include homomorphic encryption, post-quantum cryptography, parallel computing, and privacy-preserving computing.

Yu Liu is currently working toward the MS degree with the School of Computer Science, Fudan University. Her research interests include post-quantum cryptography and cryptographic protocols.

Zhe Liu received the BS and MS degrees from Shandong University, China, in 2008 and 2011, respectively, and the PhD degree from the Laboratory of Algorithmics, Cryptology and Security, University of Luxembourg, Luxembourg, in 2015. He is a professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. His research interests include security, privacy and cryptography solutions for the Internet of Things.

Yunlei Zhao received the PhD degree from Fudan University in 2004. He is now a distinguished professor with Fudan university. His main research interests include post-quantum cryptography, cryptographic protocols, theory of computing.