Phantom: A CUDA-Accelerated Word-Wise Homomorphic Encryption Library

Hao Yang[1], Shiyu Shen[2], Wangchen Dai[3], Lu Zhou[1], Zhe Liu[3], and Yunlei Zhao[2]

[1] Nanjing University of Aeronautics and Astronautics, Nanjing, China
crypto@d4rk.dev
[2] Fudan University, Shanghai, China
crypto@sher1e.dev
[3] Zhejiang Lab, Hangzhou, China
w.dai@my.cityu.edu.hk

Abstract. Homomorphic encryption (HE) is a promising technique for privacy-preserving computations, especially the word-wise HE schemes that allow batching. However, the high computational overhead hinders the deployment of HE in real-word applications. GPUs are often used to accelerate execution, but a comprehensive performance comparison of different schemes on the same platform is still missing.

In this work, we fill this gap by implementing three word-wise HE schemes BGV, BFV, and CKKS on GPU, with both theoretical and engineering optimizations. We enhance the hybrid key-switching technique, significantly reducing the computational and memory overhead. We explore several kernel fusing strategies to reuse data, resulting in reduced memory access and IO latency, and enhancing the overall performance. By comparing with the state-of-the-art works, we demonstrate the effectiveness of our implementation.

Meanwhile, we introduce a unified framework that finely integrates our implementation of the three schemes, covering almost all scheme functions and homomorphic operations. We optimize the management of pre-computation, RNS bases, and memory in the framework, to provide efficient and low-latency data access and transfer. Based on this framework, we provide a thorough benchmark of the three schemes, which can serve as a reference for scheme selection and implementation in constructing privacy-preserving applications.

Our library is available for access at `https://github.com/encryptorion-lab/phantom-fhe`. It is released under the GPLv3 license.

Keywords: Homomorphic encryption · GPU acceleration · BGV · BFV · CKKS.

1 Introduction

Homomorphic encryption (HE) is a class of cryptosystem that enables computations to be performed on encrypted data without requiring knowledge of secret keys. This allows for the construction of secure computational models that are non-interactive and do not require users to remain online throughout the evaluation process, resulting in lower communication overhead compared to other techniques like multi-party computation. HE is currently viewed as a promising building block for privacy-preserving applications such as secure neural network inference [14, 20], private set union and intersection [15], and private decision tree evaluation [36].

In 2009, Gentry introduced the concept of bootstrapping [25, 26] to bring homomorphic encryption from Somewhat Homomorphic Encryption (SHE), which requires a predetermined circuit depth, to the Fully Homomorphic Encryption (FHE) era that supports an arbitrary number of operations. Most FHE schemes currently in use are based on the (Ring-)Learning with Errors ((R-)LWE) problem [37]. These schemes can be classified as bit-wise FHE and word-wise FHE depending on the type of data and basic operations. The first class includes FHEW [21] and TFHE [18], which encrypt only a few bits per ciphertext and perform logical operations. Although they support computation of non-polynomial functions and fast bootstrapping, their ciphertext-to-message size expansion ratio is huge. Additionally, they suffer from limited message

space and parallelism, which leads to low amortized runtime on arithmetic operations such as addition and multiplication. The second class contains BGV [13], BFV [12, 22] and CKKS [17]. BGV and BFV perform exact operations on finite fields while CKKS supports approximate computations over real and complex numbers. Compared to bit-wise HE schemes, word-wise HE schemes are more efficient because they support batch processing, where multiple plaintexts can be packed into a ciphertext and evaluated in a single-instruction-multiple-data (SIMD) manner.

Many software HE libraries, including HElib [3], PALISADE [5], SEAL [41], HEEAN [1,2] and OpenFHE [4], implement RNS variants of these schemes [11, 16, 27–29] Despite the attention HE attracts, its performance is still inadequate to meet practical requirements. For instance, Brutzkus et al.'s low-latency privacy-preserving inference [14] reduces execution time for a single prediction to 2.2 seconds; however it takes over 6 hours to complete computations on the MNIST dataset [35], which only provides 10 classes and is relatively small compared to current applications such as ImageNet (1000 classes) [40].

Hardware acceleration using Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs) can help alleviate the problems mentioned above by enabling parallel execution. GPU acceleration is currently important in areas such as machine learning and has been used to speed up homomorphic encryption schemes and privacy-preserving applications in previous researches [6–8, 10, 31, 42]. They concentrate on implementing one single scheme or some HE operations and achieve speedups compared to software implementations. However, current research on accelerated homomorphic encryption schemes is still lacking in the GPU domain. Many of the latest proposed techniques have not yet been introduced into this field, and existing works only support inadequate parameter sizes with some functions that could be further optimized. Additionally, focusing on one scheme may be insufficient for real-world applications. In some cases, floating-point numbers are taken as input and approximate computation is required, while in some others precision loss may not be tolerable. These facts lead to the demand for insight into the performance of each HE scheme on the GPU under different parameter configurations. Nevertheless, it is insufficient to provide the conclusion according to current researches due to various targeted platforms and implementation approaches.

Contributions. In this work, we introduce Phantom, a GPU library for homomorphic encryption, specifically designed for high-performance, GPU-optimized implementation of three advanced word-wise HE schemes, i.e., the RNS variants of BGV [27], BFV [11, 28], and CKKS [29]. Phantom stands as the most comprehensive implementation to date, surpassing previous state-of-the-art works in terms of performance. The contributions of our work are outlined as follows:

- *Theoretical Optimizations:* We introduce several theoretical optimizations, with certain enhancements specifically tailored for GPU architecture. This includes a generalization of the Number-theoretic Weighted Transform for polynomial multiplication, achieving hierarchical implementation through arithmetic reconstruction. This approach effectively minimizes data access and IO latency for RNS polynomials across various dimensions. Additionally, we optimize the key-switching operation, reducing computational complexity, and apply the Karatsuba technique to homomorphic multiplication, thereby decreasing computational overheads. These advancements collectively contribute to reductions in both computational and memory demands in GPU-based designs.
- *Unified and Optimized GPU Implementation:* For the BGV, BFV, and CKKS schemes, we develop generalized and unified arithmetic expressions, ensuring compatibility within a single framework. This foundation supports our GPU framework, and we integrate our highly parallel GPU implementation of the schemes. We propose several GPU optimization methods, including comprehensive pre-computation, effective modulus chain and RNS base management, and a memory pool mechanism for efficient and secure data access and transfer. We also exploit diverse kernel fusing strategies and data reuse techniques to minimize data access and I/O latency, and align the implementations of the three schemes more closely.
- *Comprehensive Benchmarks:* Our framework facilitates the benchmarking of all functions of the three schemes under various parameter sets. Notably, this is the first work to comprehensively report the performance of the BGV scheme on GPU. Moreover, our implementations of the BFV and CKKS schemes outperform existing state-of-the-art works [8,31] and support larger parameter sizes, demonstrating both the efficiency and scalability of our approach.

Table 1. Summary of supported features in related works [6, 8, 10, 31] and this work. The notations are given in Sec 2.1.

Supported features		[8]	[6]	[10]	[31]	This work
Parameter	$\|N\|$	[11, 14]	[12, 16]	[13, 16]	[16, 17]	≥ 11
Schemes	BGV					✓
	BFV	✓	✓			✓
	CKKS			✓	✓	✓
Scheme functions	`PreComp`					✓
	`KeyGen`	✓				✓
	`Enc`	✓				✓
	`Dec`	✓	✓			✓
	`Ecd`					✓
	`Dcd`					✓
Operations	`HAdd`	✓		✓	✓	✓
	`CAdd`	✓		✓		✓
	`HMult`		✓	✓	✓	✓
	`CMult`			✓	✓	✓
	`HRot`				✓	✓

Comparisons with Related Works. There have been several works on accelerating word-wise HE schemes using GPUs, and the most related works are [6, 8, 10, 31, 42]. The comparisons are summarized in Table 1. In detail, the work [8] presents a GPU implementation of the BEHZ-variant BFV, focusing on `KeyGen`, `Enc`, `Dec`, `HAdd` and `HMult` with small and medium parameter sets $N \in \{2^{11}, \cdots, 2^{14}\}$. The work [6] provides both BEHZ- and HPS-variant with lager parameter sets $N \in \{2^{12}, \cdots, 2^{16}\}$, but only implements the `Dec` and `HMult` functions. Both [10] and [31] study the acceleration of CKKS and implement `HAdd`, `HMult`/`CMult` and `Rescale`, which report $N \in \{2^{13}, \cdots, 2^{16}\}$ and $\{2^{16}, 2^{17}\}$ respectively, while only [10] supports `CAdd` and only [31] supports `HRot`. The work [42] proposes a framework that accommodates three schemes, BGV, BFV and CKKS, but only provides `HMult` without `Reline`. To the best of our knowledge, there is no work so far that contains the GPU implementation of all functions of BGV, BFV and CKKS.

2 Preliminaries

2.1 Notation

Let $\mathbb{Z}$ and $\mathbb{C}$ be the group of integers and complex number, and $\mathbb{Z}_q = \mathbb{Z} \cap [-q/2, q/2)$. For an integer q and a 2-power integer N, we denote the quotient ring as $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and the corresponding residue ring modulo q as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. We use bold lower-case letters to represent ring elements (polynomials) such as $\boldsymbol{a} = \sum_{i=0}^{N-1} a_i X^i$, where a_i denote the i-th coefficient of $\boldsymbol{a}$. With a "hat" symbol such as $\hat{\boldsymbol{a}}$, we indicate that this element is in the frequency domain. The notation $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$, $\lfloor \cdot \rceil$, and $[\cdot]_q$ refer to flooring, ceiling, rounding, and modular reduction by q, respectively, which can be extended to ring elements by performing coefficient-wisely. We use $*$ to denote the convolution of two sequences and $\odot$ to denote the point-wise multiplication. For a finite set S, we use $a \xleftarrow{\$} S$ and $a \leftarrow \mathcal{X}$ to denote sampling from S uniformly or according to a distribution $\mathcal{X}$ on S, respectively.

Throughout this paper, we use `KeyGen`, `Enc`, `Dec`, `Ecd`, and `Dcd` to denote the key generation, encryption, decryption, encoding and decoding of a HE scheme, respectively. The function `PreComp` denotes the pre-computation process. The function `HAdd` (`HMult`) refers to the ciphertext-ciphertext homomorphic addition (multiplication with relinearization), `CAdd` (`CMult`) refers to the ciphertext-plaintext homomorphic addition (multiplication), and `HRot` refers to the rotation operation. The `Rescale` conducts the rescaling operation on ciphertexts.

2.2 Polynomial Multiplication

The Fourier transform ($\mathcal{F}$) and its inverse ($\mathcal{F}^{-1}$) build a bridge between operations in time and frequency domain. Namely, the convolution in one domain corresponds to the point-wise multiplication in the other domain, which can be expressed as $\mathbf{f} * \mathbf{g} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{f}) \odot \mathcal{F}(\mathbf{g}))$, where $\mathbf{f}$ and $\mathbf{g}$ are two digit sequences in time domain. Instead of complex elements, the number-theoretic transform (NTT) performs on a finite field of integers. In this work, we apply a more generic form, i.e., the Number-theoretic Weighted Transform (NWT) [19]. Formally, let ω be the primitive N-th root of unity in $\mathbb{Z}_q$ such that $\omega^N \equiv 1 \bmod q$ and $\omega^k \neq 1 \bmod q$ for all integers $0 < k < N$, and denote the weight vector $\mathbf{d} := \{d_i : d_i \neq 0, d_i \in \mathbb{Z}, i = 0, \ldots, N-1\}$ and $\mathbf{d}^{-1} = \left\{d_i^{-1} \bmod q\right\}$, the N-point forward and backward NWT of a polynomial $\boldsymbol{f} = \sum_{i=0}^{N-1} f_i X^i$ is formulated as:

$$\begin{aligned} \hat{\boldsymbol{f}} := \mathtt{NWT}_{\mathbf{d},N}(\boldsymbol{f}), \hat{f}_j &= \sum_{i=0}^{N-1} f_i d_i \omega^{ij} \pmod q \\ \boldsymbol{f} := \mathtt{INWT}_{\mathbf{d},N}(\hat{\boldsymbol{f}}), f_i &= \frac{1}{N \cdot d_i} \sum_{j=0}^{N-1} \hat{f}_j \omega^{-ij} \pmod q \end{aligned} \tag{1}$$

Indeed, it turns out that $\mathtt{NWT}_{\mathbf{d},N}(\boldsymbol{f}) = \mathtt{NTT}_N(\boldsymbol{f} \odot \mathbf{d})$ and $\mathtt{INWT}_{\mathbf{d},N}(\hat{\boldsymbol{f}}) = \mathbf{d}^{-1} \odot \mathtt{INTT}_N(\hat{\boldsymbol{f}}) \pmod q$. NTT can be viewed as a special case for $\mathbf{d} = \{1, \ldots, 1\}$, and when $\mathbf{d} = \{\psi^i : i = 0, \ldots, N-1\}$, where $\psi = \sqrt{\omega} \bmod q$, it is equivalent to the negacyclic convolution. Combined with the Fast Fourier Transform (FFT) technique, we can reduce the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

Since the multiplication of polynomials $\boldsymbol{f} = \sum_{i=0}^{N-1} f_i X^i$ and $\boldsymbol{g} = \sum_{i=0}^{N-1} g_i X^i$ produces another polynomial $\boldsymbol{h} = \sum_{i=0}^{N-1} h_i X^i$ of which the coefficients are the negacylic convolution of the sequences denoted as $\{f_i : i = 0, \ldots, n-1\}$ and $\{g_i : i = 0, \ldots, n-1\}$, this technique gives us an efficient way to compute polynomial multiplication. Throughout this paper, we perform the multiplication of two polynomials by $\boldsymbol{f} \cdot \boldsymbol{g} = \mathtt{INWT}(\mathtt{NWT}(\boldsymbol{f}) \odot \mathtt{NWT}(\boldsymbol{g}))$ with $\mathbf{d} = \{\psi^i\}$ and the FFT technique.

2.3 Basics of BGV, BFV and CKKS

In this work, we implement the RNS variants of the three schemes. Below we use BGV, BFV, and CKKS to denote for simplicity. The three schemes share several algebraic similarities but differ in some design rationales and constructions. In the following, we denote t as the plaintext modulus in BGV and BFV, the moduli chain $Q = \Pi_{i=0}^{L} q_i$ as the ciphertext modulus, and $P = \Pi_{i=0}^{k-1} p_i$ as the special modulus. During processing, we set the ciphertext modulus to Q'. For leveled schemes BGV and CKKS, $Q' := Q_\ell = \Pi_{i=0}^{\ell} q_i$ when the ciphertext is at level ℓ, $0 \leq \ell \leq L$. For scale-invariant scheme BFV, $Q' := Q$. The RNS-decomposition number is $\mathtt{dnum} := \lceil (L+1)/k \rceil$. For each ℓ, let $\alpha := k$ and $\beta := \lceil (\ell+1)/\alpha \rceil$. Below we give the specifications of the implemented schemes. A summary of important notations utilized in these schemes is provided in Table 2.

- Key generation. This module consists of the generation of public-secret key pair denoted as $(\mathsf{pk}, \mathsf{sk})$, and the key-switching keys $\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}}$.
  - Public-secret key pair. Sample $\boldsymbol{a} \xleftarrow{\$} \mathcal{R}_Q$, $\boldsymbol{s} \leftarrow \mathcal{X}$, $\boldsymbol{e} \leftarrow \mathcal{X}_e$, and compute $\boldsymbol{b} := [-\boldsymbol{a} \cdot \boldsymbol{s} + t'\boldsymbol{e}]_Q$, where $t' := t$ for BGV and $t' := 1$ for others. Set the public key as $\mathsf{pk} := (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}_Q^2$ and the secret key as $\mathsf{sk} := (1, \boldsymbol{s}) \in \mathcal{R}^2$.
  - Key-switching key. Given two secret keys $\mathsf{sk} = (1, \boldsymbol{s})$ and $\mathsf{sk}' = (1, \boldsymbol{s}')$, sample $\boldsymbol{a}'_j \xleftarrow{\$} \mathcal{R}_{PQ}$ and $\boldsymbol{e}'_j \leftarrow \mathcal{X}_e$, and compute $\boldsymbol{b}'_j = [-\boldsymbol{a}'_j \cdot \boldsymbol{s} + t'\boldsymbol{e}'_j + PB_j \cdot \boldsymbol{s}']_{PQ}$. Set the key-switching key from sk' to sk as $\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}} := \{(\boldsymbol{b}'_j, \boldsymbol{a}'_j)\}_{0 \leq j < \mathtt{dnum}} \in \mathcal{R}_{PQ}^{2 \times \mathtt{dnum}}$, where $B_j \in \mathbb{Z}_{Q_L}$ and satisfies that $B_j = 1 \pmod{q_i}$ for $j\alpha \leq i < (j+1)\alpha$ and is zero otherwise.
- Encryption. Given a public key $\mathsf{pk} = (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}_Q^2$, sample $\boldsymbol{r} \leftarrow \mathcal{X}$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \mathcal{X}_e$, compute $\mathtt{Enc}_{\mathsf{pk}}(0) := [\boldsymbol{r} \cdot (\boldsymbol{b}, \boldsymbol{a}) + t'(\boldsymbol{e}_0, \boldsymbol{e}_1)]_Q$. To encryption a plaintext $\boldsymbol{m}$, set $\boldsymbol{m}^* = [\mu \boldsymbol{m}]_t$ in BGV, $\boldsymbol{m}^* = \lfloor Q/t \rfloor [\boldsymbol{m}]_t$ in BFV, or $\boldsymbol{m}^* = \boldsymbol{m}$ in CKKS. Here, the correction factor μ is initially set to $\mu := 1$ and adjusted to $[q^{-1} \cdot \mu]_t$ upon scaling down the ciphertext by q. The resulting ciphertext is $\mathsf{ct} := \mathtt{Enc}_{\mathsf{pk}}(\boldsymbol{m}) := [\mathtt{Enc}_{\mathsf{pk}}(0) + (\boldsymbol{m}^*, 0)]_Q$.

Table 2. Summary of Notations Used in BGV, BFV, and CKKS Schemes.

Notation	Description	Notation	Description
$\mathcal{R}_{q_i}$	$\mathcal{R}_{q_i} = \mathbb{Z}_{q_i}[X]/(X^N+1)$	$\mathcal{X}$	Noise distribution
t	Plaintext modulus	L	Maximum level
Q	$Q = \Pi_{i=0}^{L} q_i$	Q_ℓ	$Q_\ell = \Pi_{i=0}^{\ell} q_i$
Q'	$Q' := Q_\ell$ or Q	P	$P = \Pi_{i=0}^{k-1} p_i$
$\mathcal{Q}_i$	$\mathcal{Q}_\ell = \{q_0, q_1, \ldots, q_i\}$	$\mathcal{P}$	$\mathcal{P} = \{p_0, p_1, \ldots, p_{k-1}\}$
$\boldsymbol{m}$	Plaintext	ct	$\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_{Q'}^2$
μ	Correction factor	δ	$\delta := P \cdot [[\mathsf{ct}']_P \cdot P^{-1}]_t$
sk	$\mathsf{sk} := (1, \boldsymbol{s}) \in \mathcal{R}^2$	pk	$\mathsf{pk} := (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}_Q^2$
q_i^*	$q_i^* = Q_\ell / q_i$	$\tilde{q}_i$	$\tilde{q}_i = [q_i^{*-1}]_{q_i}$
ν	$\nu = \left\lfloor \sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} / q_i \right\rceil$	ρ^ς	$\rho^\varsigma(X) : X \mapsto X^{5^\varsigma}$
D_j	$D_j = \Pi_{i=0}^{\alpha-1} q_{j\alpha+i}$	D^*	$D^* = \Pi_{i=\ell+1}^{\alpha\beta-1} q_i$
Q_j^*	$Q_j^* = Q_L / D_j$	$\tilde{Q}_j$	$\tilde{Q}_j = [Q_j^{*-1}]_{D_j}$
B_j	$B_j = \tilde{Q}_j Q_j^*$	$\mathtt{dnum}$	$\mathtt{dnum} := \lceil (L+1)/k \rceil$
α	$\alpha := k$	β	$\beta := \lceil (l+1)/\alpha \rceil$
$\boldsymbol{m}^*$	$\boldsymbol{m}^* = [\mu \boldsymbol{m}]_t$, $\lfloor Q/t \rfloor [\boldsymbol{m}]_t$, or $\boldsymbol{m}$		
ksk	$\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}} := \{(\boldsymbol{b}_j', \boldsymbol{a}_j')\}_{0 \le j < \mathtt{dnum}} \in \mathcal{R}_{PQ}^{2 \times \mathtt{dnum}}$		

- Decryption. Given a ciphertext $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_{Q'}^2$ and a secret key $\mathsf{sk} = (1, \boldsymbol{s})$, compute $\boldsymbol{m}' := [\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_{Q'}$. The decryption result is $[\mu^{-1}\boldsymbol{m}']$ in BGV and $\boldsymbol{m}'$ in CKKS. While in BFV, the result is $\lfloor t/Q' \cdot \boldsymbol{m}' \rceil$.
- Addition. Given two ciphertexts $\mathsf{ct}_1 = (\boldsymbol{c}_0^{(1)}, \boldsymbol{c}_1^{(1)})$ and $\mathsf{ct}_2 = (\boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(2)})$ in $\mathcal{R}_{Q'}^2$, and a plaintext $\boldsymbol{x} \in R$, the plaintext-ciphertext addition is $\mathtt{CAdd}(\mathsf{ct}_1, \boldsymbol{x}) := (\boldsymbol{c}_0^{(1)} + [\boldsymbol{x}]_{Q'}, \boldsymbol{c}_1^{(1)})$. The sum of two ciphertext is defined as $\mathtt{HAdd}(\mathsf{ct}_1, \mathsf{ct}_2) := [(\boldsymbol{c}_0^{(1)} + \boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(1)} + \boldsymbol{c}_1^{(2)})]_{Q'}$ for all three schemes. For BGV, ct_1 and ct_2 should be scaled first when the correction factors are mismatched.
- Multiplication. Given ciphertexts $\mathsf{ct}_1 = (\boldsymbol{c}_0^{(1)}, \boldsymbol{c}_1^{(1)}), \mathsf{ct}_2 = (\boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(2)}) \in \mathcal{R}_{Q'}^2$, a plaintext $\boldsymbol{x} \in R$, and the relinearization key $\mathsf{rlk} = \mathsf{ksk}_{\mathsf{sk}^2 \to \mathsf{sk}}$, the plaintext-ciphertext multiplication is defined as $\mathtt{CMult}(\mathsf{ct}_1, a) := [([\boldsymbol{x}]_{Q'} \cdot \boldsymbol{c}_0^{(1)}, [\boldsymbol{x}]_{Q'} \cdot \boldsymbol{c}_1^{(1)})]_{Q'}$, and the product of two ciphertexts is described as $\mathtt{HMult}(\mathsf{ct}_1, \mathsf{ct}_2) := [(\boldsymbol{c}_0, \boldsymbol{c}_1) + \lfloor P^{-1} \cdot \boldsymbol{c}_2 \cdot \mathsf{rlk} \rceil]_{Q'}$. Let $\mathsf{ct}' := (\boldsymbol{c}_0^{(1)} \cdot \boldsymbol{c}_0^{(2)}, \boldsymbol{c}_0^{(1)} \cdot \boldsymbol{c}_1^{(2)} + \boldsymbol{c}_1^{(1)} \cdot \boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(1)} \cdot \boldsymbol{c}_1^{(2)})$, the triple $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2)$ is defined as $[\mathsf{ct}']_{Q'}$ in BGV and CKKS, and $[\lfloor \frac{t}{Q'} \mathsf{ct}' \rceil]_{Q'}$ in BFV.
- Rotation. Given a ciphertext $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_{Q'}^2$ that encrypts a plaintext $\boldsymbol{m}(X)$, a rotation index ς and a rotation key $\mathsf{rtk}_\varsigma = \mathsf{ksk}_{\rho^\varsigma(\mathsf{sk}) \to \mathsf{sk}}$, output the encryption of $\boldsymbol{m}(\rho^\varsigma(X))$ by computing $\mathtt{HRot}(\mathsf{ct}, \varsigma) := [(\rho^\varsigma(\boldsymbol{c}_0), 0) + \rho^\varsigma(\boldsymbol{c}_1) \cdot \mathsf{rtk}_\varsigma]_{Q'}$, where the automorphism $\rho^\varsigma : \mathcal{R} \to \mathcal{R}$ is defined by $X \mapsto X^{5^\varsigma}$.

Batching. Batching is a technique that supports SIMD operations on ciphertexts by encoding multiple plaintexts into separate slots. Due to the different plaintext space, these three schemes require different mappings to the slots, i.e., $\mathcal{R}_t = \mathcal{R}/t\mathcal{R}$ in BGV and BFV and $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ in CKKS. As $X^N + 1 = \prod_{i=0}^{N-1}(X - \zeta^{2i+1}) \bmod t$ and the Chinese Remainder Theory (CRT) establishes a natural ring isomorphism from $\mathcal{R}_t = \mathbb{Z}_t/(X^N+1)$ to the product space $\mathbb{Z}_t/(X-\zeta) \times \mathbb{Z}_t/(X-\zeta^3) \times \cdots \times \mathbb{Z}_t/(X - \zeta^{2N-1}) \cong \mathbb{Z}_t^N$, we can perform N integer additions/multiplications modulo t via a single polynomial addition/multiplication in $\mathcal{R}_t$. Formally, the decoding in BGV and BFV is given by $\mathtt{BatchDcd} : \mathcal{R}_t \to \mathbb{Z}_t^N, p(X) \mapsto (p(\zeta), p(\zeta^3), \ldots, p(\zeta^{2N-1}))$ and the encoding $\mathtt{BatchEcd}$ is defined to be the inverse, which can be computed efficiently through an N-point NWT/INWT over $\mathbb{Z}_t$. While in CKKS the transformation is conducted through two steps: $\mathbb{R}[X]/(X^N+1) \xrightarrow{\sigma} \mathbb{H} \xrightarrow{\pi} \mathbb{C}^{N/2}$. First, the canonical embedding $\sigma : p(X) \mapsto (p(\xi^j))_{j \in \mathbb{Z}_{2N}^*}$ maps $p(X) \in \mathbb{R}[X]/(X^N+1)$ to $\mathbb{H} = \{(z_j)_{j \in \mathbb{Z}_{2N}^*} : z_{2N-j} = \bar{z_j}\} \subseteq \mathbb{C}^N$. Then, define T as a subgroup of the multiplicative group $\mathbb{Z}_{2N}^*$ with order $N/2$, the subring $\mathbb{H}$ can be identified with $\mathbb{C}^{N/2}$ via the natural projection $\pi : (z_j)_{j \in \mathbb{Z}_{2N}^*} \mapsto (z_j)_{j \in T}$. Through this we have the encoding and

decoding functions in CKKS, i.e., $\texttt{CKKSEcd}(z \in \mathbb{C}^{N/2}; \Delta) : \mathbb{C}^{N/2} \to \mathcal{R}, z \mapsto m = \lfloor \Delta \cdot \sigma^{-1}(\pi^{-1}(z)) \rceil$ and $\texttt{CKKSDcd}(m \in \mathcal{R}; \Delta) : \mathcal{R} \to \mathbb{C}^{N/2}, m \mapsto z = \pi \circ \sigma(\Delta^{-1} \cdot m)$, respectively. Here, the σ map is performed through FFT and σ^{-1} is the inverse.

RNS Representation. The residue number system (RNS) is often applied to handle the computation of elements larger than machine word-size. With the RNS base $\{q_0, q_1, \ldots, q_\ell\}$, a polynomial $\boldsymbol{f} \in \mathcal{R}_{Q_\ell}$ can be represented as $(\boldsymbol{f}_0, \ldots, \boldsymbol{f}_\ell) \in \Pi_{i=0}^{\ell} \mathcal{R}_{q_i}$, and multi-precision arithmetic can be replaced by a set of residues-wise arithmetic through the isomorphism $\mathcal{R}_{Q_\ell} \to \mathcal{R}_{q_0} \times \mathcal{R}_{q_1} \times \cdots \times \mathcal{R}_{q_\ell}$. The RNS-friendly feature of BGV and CKKS allows double-CRT representation of ciphertexts most of the time, i.e., $(\hat{\boldsymbol{f}}_0, \ldots, \hat{\boldsymbol{f}}_\ell) \in \Pi_{i=0}^{\ell} \mathcal{R}_{q_i}$ in frequency domain. For BFV that additional RNS base conversions are needed to solve the compatibility of some operations, it is better to store ciphertexts in the RNS representation.

Modulus-Switching. We define the modulus-switching as switching the modulus (equivalently, RNS base) of a ring element from one to another. Two methods are commonly utilized for fast RNS base conversion from $\mathcal{Q}_\ell = \{q_0, q_1, \ldots, q_\ell\}$ to $\mathcal{B} = \{r_0, r_1, \ldots, r_{l-1}\}$, i.e., the BEHZ-type [11] and the HPS-type [28], which are illustrated as:

$$\begin{aligned} \texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}_\ell \to \mathcal{B}}(x) &= \left(\left[\sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* \right]_{r_j} \right)_{j=0}^{l-1} \\ \texttt{Conv}^{\text{HPS}}_{\mathcal{Q}_\ell \to \mathcal{B}}(x) &= \left(\left[\sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* - \nu Q_\ell \right]_{r_j} \right)_{j=0}^{l-1} \end{aligned} \tag{2}$$

Here, $\nu = \left\lfloor \sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} / q_i \right\rceil$, $q_i^* = Q_\ell / q_i$, and $\tilde{q}_i = [q_i^{*-1}]_{q_i}$. The method to extend and reduce the RNS base are based on this technique, defined as $\texttt{ModUp}_{\mathcal{Q}_\ell \to \mathcal{Q}_\ell \cup \mathcal{B}}([\boldsymbol{c}]_{\mathcal{Q}_\ell}) := ([\boldsymbol{c}]_{\mathcal{Q}_\ell}, \texttt{Conv}_{\mathcal{Q}_\ell \to \mathcal{B}}([\boldsymbol{c}]_{\mathcal{Q}_\ell}))$ and $\texttt{ModDown}_{\mathcal{Q}_\ell \cup \mathcal{B} \to \mathcal{Q}_\ell}([\boldsymbol{c}]_{\mathcal{Q}_\ell}, [\boldsymbol{c}']_{\mathcal{B}}) := ([\boldsymbol{c}]_{\mathcal{Q}_\ell} - \texttt{Conv}_{\mathcal{B} \to \mathcal{Q}_\ell}([\boldsymbol{c}']_{\mathcal{B}})) \cdot [R^{-1}]_{\mathcal{Q}_\ell}$, where $R = \Pi_{j=0}^{l-1} r_j$. For a given ciphertext $\mathsf{ct} \in \mathcal{R}^2_{Q_\ell}$, the `Rescale` operation scales down the modulus of ct from Q_ℓ to $Q_{\ell-1}$, by computing $\texttt{ModDown}_{\mathcal{Q}_\ell \to \mathcal{Q}_{\ell-1}}(\mathsf{ct})$.

For the BGV scheme, the `ModDown` is slightly distinct, which necessitates a correction to ensure accuracy. Given that it involves a plaintext modulus t, the conversion of ct from $\mathcal{R}_{QR}$ to $\mathsf{ct}' \in \mathcal{R}_Q$ must satisfy the condition $\mathsf{ct}' \equiv \mathsf{ct} \pmod{t}$. Denoting $\mathsf{ct}_s = \lfloor \mathsf{ct}/R \rceil$ as the item after scaling, a rounding error $\epsilon = \mathsf{ct} - R\mathsf{ct}_s$ is observed. Consequently,

$$\mathsf{ct}' = (\mathsf{ct} - \epsilon + R \cdot [\epsilon \cdot R^{-1}]_t) \cdot R^{-1} \tag{3}$$

Since $\epsilon = \mathsf{ct} - R\mathsf{ct}_s$, it follows that $\epsilon = [\mathsf{ct}]_R$. Therefore, in the $\texttt{ModDown}_{\mathcal{Q}_\ell \cup \mathcal{B} \to \mathcal{Q}_\ell}$ for BGV, it is imperative to add the term $\delta := R \cdot [[\mathsf{ct}]_R \cdot R^{-1}]_t$ to the ciphertext prior to performing the final multiplication by R^{-1}. Meanwhile, the correction factor μ should be updated as $[R^{-1} \cdot \mu]_t$.

Key-Switching. We implement an optimized key-switching procedure with the HPS technique [28]. Given a key-switching key $\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}}$ and a ciphertext $\mathsf{ct}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1) \in \mathcal{R}^2_{Q'}$ under $\mathsf{sk}' = (1, \boldsymbol{s}')$, this procedure splits $\boldsymbol{c}'_1$ into β digits with base $\mathcal{D}'_j = \{q_{j\alpha}, \ldots, q_{(j+1)\alpha-1}\}$ for $j \in [0, \beta - 1)$ and $\mathcal{D}'_{\beta-1} = \{q_{\alpha(\beta-1)}, \ldots, q_\ell\}$, then it raises the base of each digit to $\mathcal{Q}' \cup \mathcal{P}$, and computes the ciphertext $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}^2_{Q'}$ by $\mathsf{ct} = \left[(\boldsymbol{c}'_0, 0) + \left\lfloor P^{-1} \cdot [\sum_{j=0}^{\beta-1} c'_{1,j} \cdot \mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}, j}]_{PQ'} \right\rceil \right]_{Q'}$. Through this process, the ciphertext ct' is transformed into ct that encrypts an approximately equivalent message using another key $\mathsf{sk} = (1, \boldsymbol{s})$, i.e., $\langle \mathsf{ct}, \mathsf{sk} \rangle = \langle \mathsf{ct}', \mathsf{sk}' \rangle + \boldsymbol{e}_{\mathsf{ks}}$, where $\boldsymbol{e}_{\mathsf{ks}}$ is the noise.

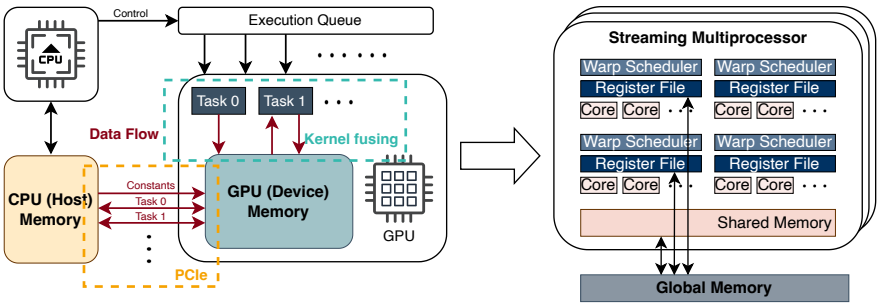

Fig. 1. The CPU-GPU computational model and the architecture of GPU.

2.4 GPU Programming

We summarize the computational model and the architecture of GPU in Figure 1. The GPU memory can be classified into two types. The first is the read-write memory, including the global memory (GMEM), the shared memory (SMEM), and the register file (RF), where the access speed is from slow to fast as listed. The second is the read-only memory, including the constant memory and the texture memory, and both of them can be cached. A CUDA kennel is run concurrently on GPU by many threads, which is the minimum execution unit and can be grouped into a block. Every thread has its private RF, and the SMEM is shared by all threads within a block. The GMEM and read-only memory are accessible for all threads, which have the longest lifetime that encompass the entire computational task. During execution, threads are bundled per 32 in a warp. A streaming multiprocessor (SM) holds one or multiple blocks, and each warp scheduler (WS) in the SM schedules and executes one warp at a time. In a heterogeneous platform that equipped with CPU and GPU, a common and straightforward collaborative computing mechanism is that the CPU schedules the tasks in the execution queue, launches the corresponding kernels to the GPU and transfers essential data through PCIe, and then waits for the GPU to execute and return the results. In this case, fusing data-dependent kernels can reduce the IO latency caused by data transfer and memory access to some extent. For better performance, over-fusing should be prevented, as the SM occupancy will decrease if the resource consumption of a block is too high.

3 Implementation Details and Optimizations

In this section, we present the implementation details of Phantom. First, we develop a framework for implementing and benchmarking the three HE schemes. The architecture of the framework is summarized in Sec 3.1. Then, we describe the design of essential modules from the low-level operations to the high-level schemes, covering the implementation and optimization of kernels, and methods to adapt the three schemes.

3.1 Framework Structure of Phantom

We show the structure of the implemented framework in Figure 2 to give a concise overview. In functionality, it consists of two main parts, one serving for pre-computation and the other containing optimized implementation of the HE schemes that can be divided into three layers: a math/polynomial layer, a RNS arithmetic layer and a scheme layer.

The basic layer contains the low-level modular operations for both 64-bit and 128-bit integers, a pseudo-random generator and polynomial arithmetics. For the integer operations, we provide well-optimized constant-time implementation and minimize the number of machine instructions and register usage. Based on this, we implement polynomial arithmetic, such as NWT and FFT for fast and low-complexity computation. At the middle layer, we implement the sampling and RNS arithmetic modules, of which the operands are polynomials under RNS or double-CRT representation. The sampling module consists of three approaches for sampling

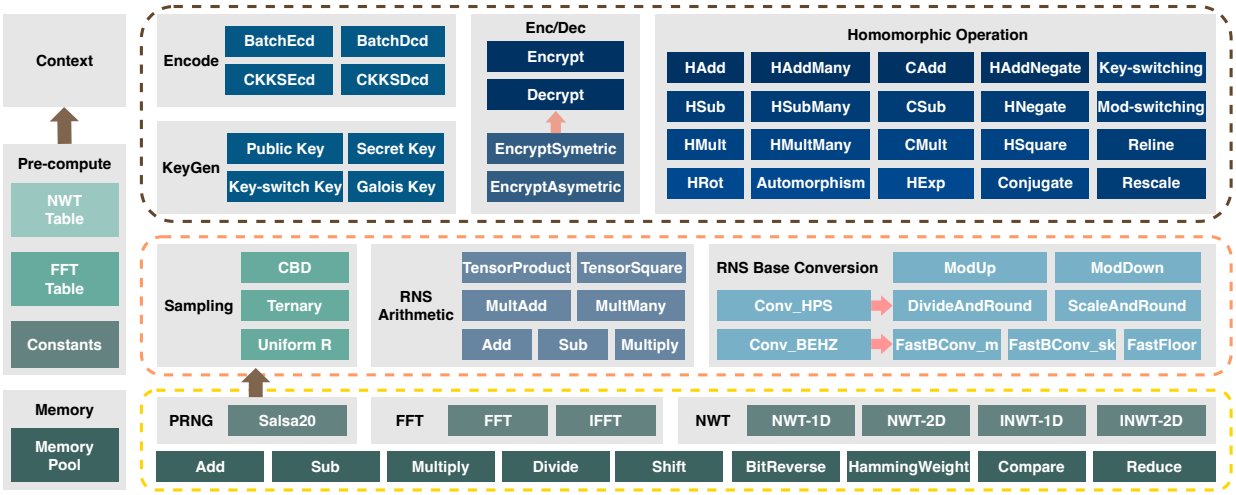

Fig. 2. Structure of our benchmarking framework. The implemented functions are organized into three layers, i.e., a math/polynomial layer, an RNS arithmetic layer and a scheme layer.

polynomial coefficients from ternary, uniform and centered binomial distribution. The RNS module offers efficient polynomial arithmetics, and we implement common algorithms of both BEHZ- and HPS-type base conversion. The top layer offers high-level unified implementation of the BGV, BFV and CKKS schemes. For all schemes, our framework supports the following features: (1) both symmetric and asymmetric encryption; (2) homomorphic addition, subtraction and multiplication of two or multiple plaintexts and ciphertexts; (3) in-place homomorphic exponentiation, negation, and rotation of a single ciphertext.

Our optimizations are primarily targeted towards memory efficiency in addressing the significant memory consumption and data access demands of HE schemes. Firstly, we optimize the data access patterns to eliminate stride GMEM access, ensuring that warp-level memory requests are executed within a single cycle for increased efficiency. Secondly, we introduce a memory pool mechanism for the efficient and secure handling and transfer of large data volumes. Thirdly, our approach includes the development of various kernel fusion techniques. This encompasses both intra-kernel fusion, which consolidates arithmetic operations within a single kernel allowing for the reuse of temporary elements in registers, and inter-kernel fusion, which amalgamates adjacent kernels sharing similar parallelism. This latter technique is designed to maintain temporary data in lower latency memory, thereby reducing the need for time-consuming GMEM transfers. Overall, these enhancements contribute to a reduction in memory request instructions and IO latency, thus significantly improving the performance of our HE implementations.

3.2 Number-theoretic Weighted Transform

We implement the NWT using two distinct methodologies, aiming at maximizing parallelism and minimizing total IO latency within the hierarchical memory structure of GPUs across various input data sizes.

NWT-1D. For the NWT of a polynomial $\boldsymbol{f} \in \mathcal{R}_{Q_\ell}$ in RNS representation, a widely applied approach [8,39] involves deploying a single kernel where each block performs residue-wise transform under q_i. In a block, each thread is responsible for loading ι coefficients and executing a radix-ι butterfly operation. For larger N, this kernel is called recursively to process polynomial segments in cases of limited memory. However, this approach encounters challenges with increasing N. Firstly, a single kernel results in an exponential increase in per-thread register usage, causing register overuse and a decline in SM occupancy. Secondly, decreasing register usage per thread requires to reduce coefficients loaded per thread and butterfly radix, leading to increased register-memory interaction and IO latency, especially problematic when SMEM capacity is insufficient for

large N. Lastly, while recursive processing is feasible, it is not generic and necessitates multiple kernels, leading to excessive GMEM access.

NWT-2D. To address these challenges, we apply arithmetic reconstruction, aiming to divide the process into as few kernels as possible, thus minimizes overall GMEM access and reduces overhead of each kernel. This approach follows a hierarchical methodology [31, 34, 42]. Here, we divide the N-point NWT into two stages using suitable N_1 and N_2 where $N = N_1 N_2$. Let $\psi^{2N} \equiv 1 \bmod q$, we define $\texttt{NWT}_{\mathbf{d},N}$ and $\texttt{INWT}_{\mathbf{d},N}$ with $\mathbf{d} = \{\psi^i : i = 0, \ldots, N-1\}$ as follows:

$$\begin{aligned}
\hat{x}_{y+N_1 z} &= \sum_{j=0}^{N_2-1} \sum_{i=0}^{N_1-1} x_{iN_2+j} \psi_N^{2(iN_2+j)(y+N_1 z)+(iN_2+j)} \\
&= \sum_{j=0}^{N_2-1} \left(\psi_N^{2jy+j} \cdot \left(\sum_{i=0}^{N_1-1} x_{iN_2+j} \psi_{N_1}^{2iy+i} \right) \right) \psi_{N_2}^{2jz} \bmod q \\
x_{iN_2+j} &= \frac{1}{N_1 N_2} \sum_{z=0}^{N_2-1} \sum_{y=0}^{N_1-1} \hat{x}_{y+N_1 z} \psi_N^{-2(iN_2+j)(y+N_1 z)-(iN_2+j)} \\
&= \sum_{y=0}^{N_1-1} \frac{\psi_{N_1}^{-2iy-i}}{N_1} \left(\frac{\psi_N^{-2jy-j}}{N_2} \left(\sum_{z=0}^{N_2-1} \hat{x}_{y+N_1 z} \psi_{N_2}^{-2jz} \right) \right) \bmod q
\end{aligned} \tag{4}$$

This pattern enables us to implement two kernels, each responsible for one of the sub-procedures. Actually, the one kernel `NWT-1D` is equivalent to one phase in `NWT-2D`. This method significantly reduces the total GMEM access and IO latency, as GMEM access is required only between the two kernels.

Detail Instantiation. The `NWT-1D` method exhibits high hardware consumption for large N, but is sufficiently memory-efficient for small N, since it obviates the need for dual kernels and GMEM interactions. However, given that HE parameters often require large N values, with $|N| \geq 12$, we opt for the `NWT-2D` method, which strikes a balance between performance and memory usage. We incorporate `NWT-1D` with $\iota = 8$ as the building block for the two kernels in `NWT-2D`. For the (N_1, N_2) configuration, we set $|N_1| = 6$ for $|N| = 12$, $|N_1| = 7$ for $|N| = 13$, and $|N_1| = 8$ for $|N| \in [14, 17]$. This setup minimizes SMEM interactions and achieves a near-equal split between N_1 and N_2, which provides better theoretical complexity. In each kernel, multiple RNS polynomials under different moduli are batched. Unlike previous works, where both [34] and [31] only support $|N| \in [14, 17]$, and [42] supports $|N| \in [11, 17]$ but fixes $N_2 = 2^{11}$, our implementation can handle polynomials of $|N| \leq 17$ and is more flexible.

Optimization in INWT. The inverse transformation involves handling the division by N. Prior works [31, 34, 42] adopt a divide-by-2 strategy and integrate this division into the pre-compute table of ψ. However, this method poses limitations with the GS butterfly [24, 30], as it affects only half of the coefficients, necessitating additional steps for the remaining half. To address this, we retain the divide-by-N operation in the final level of `INWT` process. This approach ensures a more balanced processing between the left and right branches of the butterfly algorithm. Furthermore, we can fuse the division operation with other processes to reduce computational overhead, such as scaling in modulus-switching.

3.3 RNS Base Management

We implement an RNS base management module to provide direct access and conversion of ciphertext moduli, of which the structure is shown in Figure 3. This module consists of two parts. The first part provides access to the RNS bases at each level, including the ciphertext base $\mathcal{Q}$, the base $\mathcal{P}$ for key-switching, and the auxiliary base. The second part is implemented to offer pre-computed values for base conversions used in operations such as key-switching and division-and-rounding. The instantiation is according to the pre-set base conversion technique, i.e., BEHZ [11] or HPS [28].

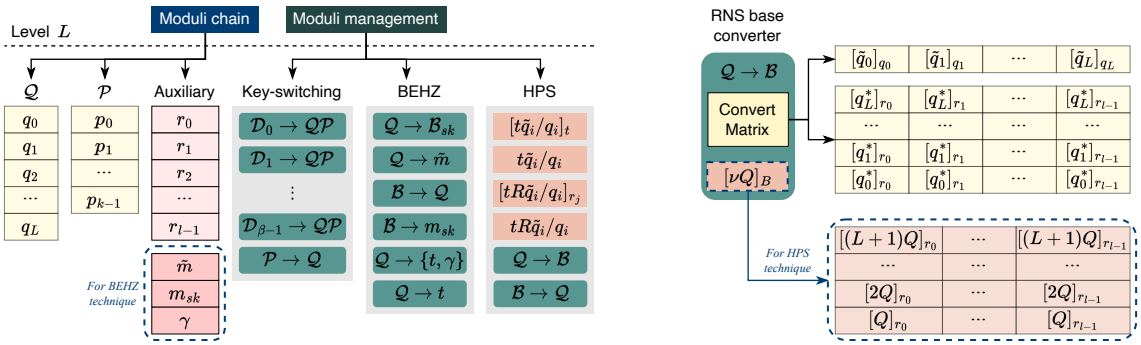

Fig. 3. Design of the RNS base management module and base converter in our framework. The auxiliary base and the base converter are initialized depending on the conversion type of BEHZ or HPS.

Modulus-switching. To switch the RNS base of an element $\boldsymbol{x}$ from base $\mathcal{Q}_\ell = \{q_0, q_1, \ldots, q_\ell\}$ to $\mathcal{R} = \{r_0, r_1, \ldots, r_{l-1}\}$, we implement both $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}_\ell \to \mathcal{R}}(\boldsymbol{x})$ and $\texttt{Conv}^{\text{HPS}}_{\mathcal{Q}_\ell \to \mathcal{R}}(\boldsymbol{x})$. Since the computational flow has different dimensions for parallel execution, i.e., $(\ell+1)$-parallelism or l-parallelism, implementing one kernel and setting the parallelism directly to one of the values can lead to some computations being repeatedly executed. Based on this consideration, we implement two kernels for the RNS base conversion. In the first kernel, we compute $[x_i \cdot \tilde{q}_i]_{q_i}$ and utilize $(\ell+1) \cdot N$ threads. In the second kernel, we launch $l \cdot N$ threads to compute the modular multiplication with q_i^* and the accumulation. We store the accumulated values in the RF and reduce the number of modular reductions by lazy reduction. Each thread takes a polynomial coefficient and multiplies it with the corresponding elements of the conversion matrix, which are pre-computed and provided by the base converter. For HPS-type conversion, we fused the process of subtracting $[\nu Q]_R$ into the second kernel. In this case, each thread also needs to compute ν as an index to obtain the value of $[\nu Q]_R$ from the converter and subtract it from the result.

3.4 Key-Switching

The hybrid key-switching technique is first proposed for CKKS in [29] and then introduced into BGV and BFV in [33]. It is the most practical method, which has been demonstrated in the work [31] on accelerating CKKS with GPUs. In our implementation, we first utilize two techniques to obtain a more efficient key-switching procedure. Then, we give a generic design compatible with the three schemes and methods to optimize the memory usage.

Optimized Key-Switching Procedure. We first recall the original design in [29, 31]. Denote $D_j = \Pi_{i=0}^{\alpha-1} q_{j\alpha+i}$, $D^* = \Pi_{i=\ell+1}^{\alpha\beta-1} q_i$, $Q_j^* = Q_L/D_j$, and $\tilde{Q}_j = [Q_j^{*-1}]_{D_j}$. Given the ciphertext $\mathsf{ct}' = (\boldsymbol{c}_0', \boldsymbol{c}_1') \in \mathcal{R}^2_{Q_\ell}$ under $\mathsf{sk}' = (1, \boldsymbol{s}')$, this procedure first raises the modulus of $\boldsymbol{c}_1'$ to the nearest $\Pi_{j=0}^{\beta-1} D_j$ by multiplying D^*, such that $\Pi_{j=0}^{\beta-2} D_j < Q_\ell \le \Pi_{j=0}^{\beta-1} D_j$. Then, the polynomial is divided into β digits. Afterwards, it raises the modulus of each digit to $Q_{\alpha\beta-1}P$ to multiply with the $\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}}$, and at last reduce the modulus of the accumulation result to Q_ℓ. In our implementation, we eliminate the `Modup` of $\boldsymbol{c}_1'$, i.e., we keep the base of $\boldsymbol{c}_1'$ in $\mathcal{Q}_\ell$ and extend the base of each digit to $\mathcal{Q}_\ell\mathcal{P}$. This saves $\alpha\beta - \ell - 1$ (`I`)`NWT`s and point-wise multiplications of polynomials. Additionally, we apply the HPS technique [28] by setting $B_j = \tilde{Q}_j Q_j^*$ in the generation of key-switching keys, $j \in [0, \beta)$, as illustrated in Sec 2.3. With this optimization, we do not need to perform the scalar multiplication with $\tilde{Q}_j$ and the RNS decomposition is executed implicitly without additional consumption.

Generic Design. The homomorphic multiplication and rotation over ciphertexts both change the underlying secret key implicitly, but differ in the input formats. Meanwhile, the ciphertexts of the three schemes are

kept in different representations. This motivates us to explore a generic design that is compatible with all HE operations of the three schemes. In detail, the homomorphic multiplication yields a triple $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2)$ under secret key $(1, \boldsymbol{s}^2)$, and the automorphism ρ^ς applied in rotation produces $(\rho^\varsigma(\boldsymbol{c}_0), \rho^\varsigma(\boldsymbol{c}_1))$ under $(1, \rho^\varsigma(\boldsymbol{s}))$. Thus, we unify the input of key-switching module to $(\boldsymbol{g}_0, \boldsymbol{g}_1, \boldsymbol{g}_2)$, and instantiate it as $(\boldsymbol{c}_0, 0, \boldsymbol{c}_1)$ in normal switching case and $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2)$ in relinearization. Figure 5 shows the details of our design. We adapt the entire procedure to three steps. First, for the schemes BGV and CKKS that keep the ciphertexts in the double-CRT representation, we transform them to the normal domain before `ModUp` by `INWT`. Then, we perform the same computational sequences for all three schemes, i.e. digit-wise `ModUp`, `NWT`, inner product, and `INWT`. At last, we apply three branches to handle different cases of the three schemes, containing the ciphertexts representation, the multiplication of P^{-1}, and the correction in `ModDown` for BGV.

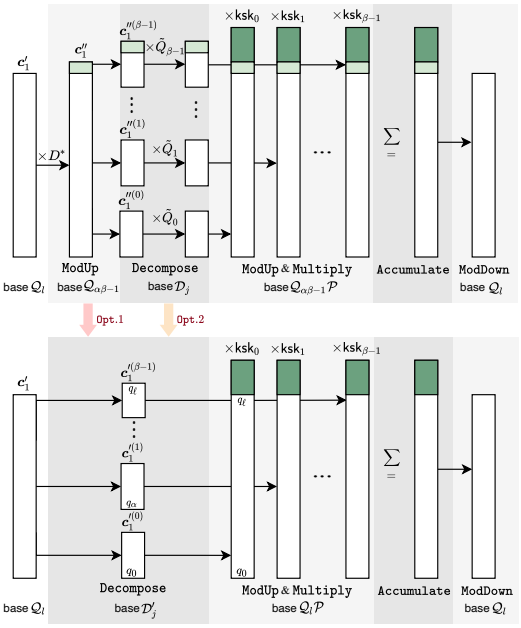

Fig. 4. The computational flow of optimized key-switching procedure.

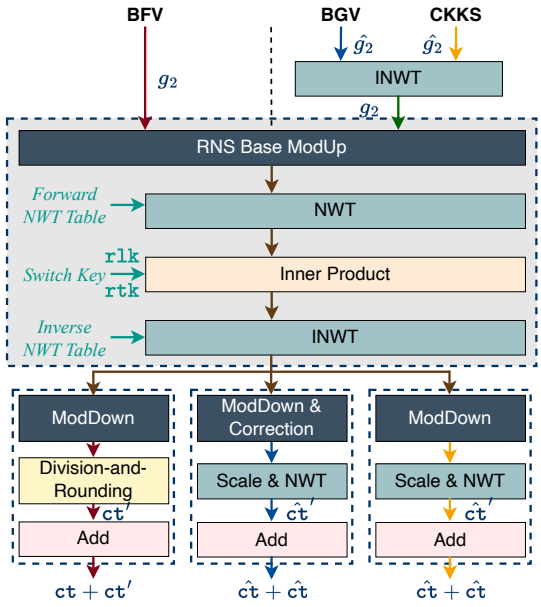

Fig. 5. The generic design of the key-switching module for three schemes. We use the symbol "&" to mark the fused operation in our implementation.

Kernel Fusing. We introduce three specific types of kernel fusing in key-switching aimed at optimizing data access patterns. Firstly, for the `ModUp` operation, we reorganize the storage sequence of the original digits and conversion results. This rearrangement enables the fusing of `cudaMemcpy` operations for base conversion results and digits into a single, streamlined process, thereby simplifying the computation and reducing IO latency in data copying. Secondly, in BGV scheme, the `ModDown` operation necessitates a term $\delta := P \cdot [[\mathsf{ct}']_P \cdot P^{-1}]_t$ for correction, as described in (3). We perform an intra-fusing strategy by integrating this step with the residue-wise subtraction in base conversion, capitalizing on their identical parallelism and shared modulus q_i. Thirdly, considering that both BGV and CKKS schemes require a scaling operation adjacent to the NWT, we conduct an inter-fusing approach by amalgamating this scaling operation within the INWT kernels for these schemes. These intra-fusing and inter-fusing approaches allow for the retention of temporary data and most intermediate values between the two processes within the RF, enabling direct processing and eliminating the need for supplementary data transfers, thus ensuring minimal IO latency. As a result, all these techniques effectively reduce the overall data access demands, thereby enhancing the operational efficiency of these HE schemes.

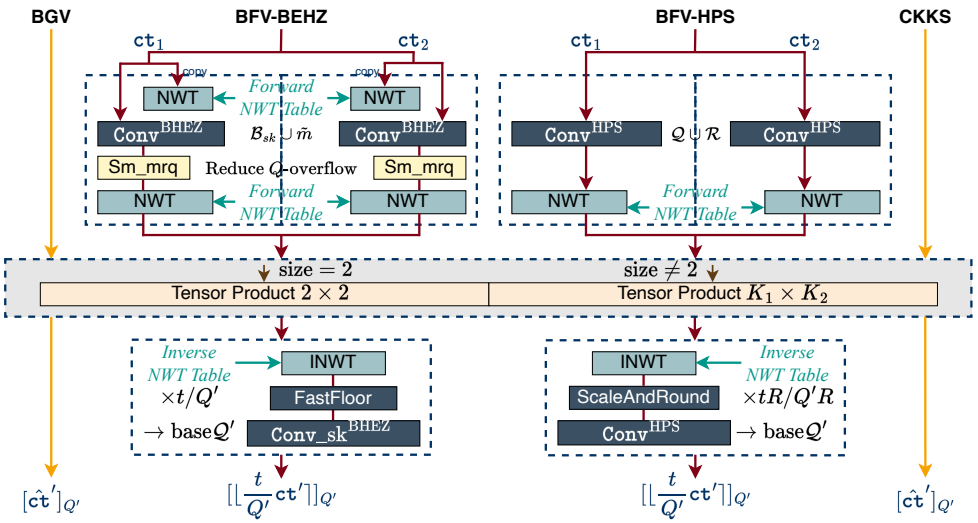

Fig. 6. Homomorphic multiplication for three schemes. We apply four branches to make the module compatible with BGV, CKKS, and two variants of BFV.

3.5 Homomorphic Multiplication

The multiplication of two given ciphertexts $\mathsf{ct}_1 = (\boldsymbol{c}_0^{(1)}, \boldsymbol{c}_1^{(1)}), \mathsf{ct}_2 = (\boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(2)}) \in \mathcal{R}_{Q'}$ consists of two phases of tensor product and relinearization. Figure 6 gives the concise design for the first phase, in which we obtain the triple $\mathsf{ct}' = (\boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2) := (\boldsymbol{c}_0^{(1)} \cdot \boldsymbol{c}_0^{(2)}, \boldsymbol{c}_0^{(1)} \cdot \boldsymbol{c}_1^{(2)} + \boldsymbol{c}_1^{(1)} \cdot \boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(1)} \cdot \boldsymbol{c}_1^{(2)})$. For BGV and CKKS, since we keep the ciphertext under double-CRT representation, there is no need for NWT/INWT transformation before and after the tensor product compared to BFV of which the ciphertexts are under RNS representation. The main reason for this is that BFV needs the scaling $[\lfloor \frac{t}{Q'} \mathsf{ct}' \rceil]_{Q'}$, which the BEHZ- [11] and HPS-variant [28] adopt different methods to adapt.

In the tensor product kernel, we consider all possible circumstances, depending on the size of the input ciphertexts. The default and more recommended case is that every ciphertext contains two ring elements, since a large size leads to more consumption for computation and memory. The common method to compute $\hat{\boldsymbol{d}}_1$ is implemented by $\hat{\boldsymbol{d}}_1 = [\hat{\boldsymbol{c}}_0^{(1)} \odot \hat{\boldsymbol{c}}_1^{(2)}]_{Q'} + [\hat{\boldsymbol{c}}_1^{(1)} \odot \hat{\boldsymbol{c}}_0^{(2)}]_{Q'}$. In our implementation, we apply the Karatsuba technique [32] by first computing $\hat{\boldsymbol{d}}_1' = [(\hat{\boldsymbol{c}}_0^{(1)} + \hat{\boldsymbol{c}}_1^{(1)}) \odot (\hat{\boldsymbol{c}}_0^{(2)} + \hat{\boldsymbol{c}}_1^{(2)})]_{Q'}$, and then obtain $\hat{\boldsymbol{d}}_1$ through $\hat{\boldsymbol{d}}_1 = [\hat{\boldsymbol{d}}_1' - \hat{\boldsymbol{d}}_0 - \hat{\boldsymbol{d}}_2]_{Q'}$. As the 128-bit product result is obtained through two 64-bit CUDA PTX instructions, the eliminated point-wise multiplication by Karatsuba technique actually trades two 64-bit multiplication instructions by cheaper addition and shift-right instructions per coefficient.

BFV variants. To solve the compatibility of division-and-rounding with RNS, both BEHZ- and HPS-variant require an auxiliary base, denoting as $\mathcal{B}_{sk} \cup \tilde{m}$ and $\mathcal{R}$ respectively, as well as the modulus-switching to the extended base and to the original base $\mathcal{Q}'$ before and after the tensor product. Since the most time-consuming parts are the NWT/INWT and the tensor product, the overall difference in computational overhead between these two variants is not significant. However, the memory consumption is different. The BEHZ-variant requires three base converters, including $\mathcal{Q}' \to \mathcal{B}_{sk} \cup \tilde{m}$, $\mathcal{Q}' \to \mathcal{B}_{sk}$, and $\mathcal{B}_{sk} \to \mathcal{Q}'$, while the HPS-variant needs $\mathcal{Q}' \to \mathcal{B}$ and $\mathcal{B} \to \mathcal{Q}'$. For base conversion $\mathcal{Q} \to \mathcal{B}$, each converter pre-computes $\{[\tilde{q}_i]_{q_i}, [q_i^*]_{r_j}\}$ for BEHZ-variant and $\{[\tilde{q}_i]_{q_i}, [q_i^*]_{r_j}, [\nu Q']_{r_j}\}$ for HPS-variant. Additionally, the HPS-variant also needs pre-computation of $[tR\tilde{q}_i/q_i]_{r_j}$ and $tR\tilde{q}_i/q_i$.

4 Performance Evaluation and Comparison

In this section, we present the performance of the three HE schemes and the comparisons with related works.

4.1 Experiment Setup

We compile the implementations using g++ 11.4 and CUDA 12.2 on Ubuntu Server 22.04. CPU experiments are performed on an Intel(R) Core(TM) i9-12900KS CPU with 16 cores. GPU experiments are performed on NVIDIA Tesla A100 80G PCIe GPU. The performance results are the median of 100 tests.

Table 3. Maximum Ciphertext Modulus Size $|QP|$ Corresponding to Various Security Levels

$\|N\|$	11	12	13	14	15	16	17
$\lambda = 80$	86	173	348	704	1428	2889	5868
$\lambda = 128$	54	108	218	438	881	1777	3576
$\lambda = 192$	37	75	151	304	611	1229	2469

All parameter sets that we used to evaluate the performance of schemes achieve 128 bit security, except in testing the key-switching procedure (in Table 5), for maintaining the same parameter configuration with [31]. In Table 3, we detail the maximum size of the ciphertext modulus $|QP|$ corresponding to security levels of 80, 128, and 192 bits, as calculated using the LWE estimator [9]. The choice of security parameters influences performance, particularly because the selection of the modulus chain. When the number of moduli in QP increases, specifically q_i and p_i, there may be a decrease in the security level but an allowance for a deeper multiplication depth. Concurrently, the residue-wise computation overhead escalates, leading to an overall increase in execution time.

The CPU baseline is obtained from SEAL benchmarking [41] with the default parameter sets. We compare our GPU implementation with the works [8, 31, 38]. Currently, there is no complete GPU implementation of BGV. The works [8, 31] are the state-of-the-art implementations of BFV and CKKS, respectively, and the work [38] present a latest GPU implementation of BFV. The work [31] open-sources some code of the lower-level operations of CKKS, and we run their code on our GPU to compare the performance. For the performance of the high-level homomorphic operations that are closed-source, we used the data provided in their papers. In [23], the authors present a GPU implementation of CKKS using Tensor Core Units. We do not compare our implementation with this work because they implement 32-bit arithmetic, which leads to significant speedup since [31] focuses on 64-bit arithmetic.

4.2 Comparison with State-of-the-Art Work

Our comparisons focus on (I)NWT and key-switching implementations against [31]. These operations are crucial in HE and often represent significant computational bottlenecks, which also encompass almost all low-level implementation.

In Table 4, we present an analysis of this work (abbreviated as TW) and [31], which includes single (I)NWT executions under identical modulus and same platform. The metrics include execution time and DRAM bandwidth. Due to our optimized and constant-time low-level arithmetic implementation, our implementation outperforms [31] across all $|N| \in [12, 17]$, both in speed and memory efficiency. Specifically, for NWT, we achieve improvements of up to 8.3% in speed and 9.1% in memory throughput. For INWT, our divide-by-N optimization yields even greater enhancements, with speedups of up to 19.8% and a 24.7% increase in memory throughput.

Table 5 compares the performance of key-switching module. Because the authors do not release the complete key-switching implementation, we compare the three individual steps: `ModUp`, `Product`, and `ModDown`, and `ModDown`, and the total time is estimated from these. The released code [31] contains pure kernel implementation, omitting additional processing such as memory and scheme management, which, though adding

Table 4. Comparison of (I)NWT implementation in this work (TW.) and related work [31] on Tesla A100 GPU.

	NWT				INWT			
	Execution time (μs)		Bandwidth (GB/s)		Execution time (μs)		Bandwidth (GB/s)	
$\|N\|$	[31]	TW.	[31]	TW.	[31]	TW.	[31]	TW.
12	-	18.87	-	3.47	-	17.34	-	3.78
13	-	20.16	-	6.50	-	17.91	-	7.32
14	22.18	20.64	11.82	12.70	23.54	19.25	11.14	13.62
15	24.30	22.28	21.58	23.54	24.94	20.01	21.02	26.21
16	24.08	22.82	43.55	45.96	26.02	21.38	40.30	49.05
17	26.04	25.51	80.54	82.20	28.28	27.16	74.15	77.22

Table 5. Performance break down of the hybrid key-switching module of our CKKS implementation and [31] on Tesla A100 GPU. The functions `ModUp`, `Product`, and `ModDown` denote the modulus-raising, inner product, and modulus-down, respectively.

	Execution time (μs)								
	[31]	TW.	[31]	TW.					
$\|N\|$	15	15	16	16	16	16	16	16	16
$\|Q\|$	1,510	1,510	2,260	2,260	2,260	2,260	2,260	2,260	2,260
$\|QP\|$	2,410	2,410	3,160	3,160	3,160	3,160	3,160	3,160	3,160
ℓ	15-29	15	30-44	44	42	39	36	33	30
k	15	15	15	15	15	15	15	15	15
`dnum`	2	2	3	3	3	3	3	3	3
`ModUp`	253	169	874	938	887	818	769	700	648
`Product`	76	55	244	236	228	217	206	196	185
`ModDown`	144	106	324	348	336	317	300	283	265
Total	**617**	**436** (**1.4**×)	**1,766**	1,870	1,787	1,669	1,575	1,462	**1,363** (**1.3**×)

some overhead in our case, are necessary for practical use. Despite this, our implementation demonstrates superior performance as ciphertext level decreases. In contrast, the execution time in [31] remains constant, as the raised modulus is unchanged. Our method shows a gradual reduction in total time with decreasing modulus levels, enabling a key-switching speedup of more than 1.3×.

4.3 Effectiveness of Memory Optimizations

In Fig. 7, we present the execution times before and after implementing our memory optimization strategies. These results correspond to the three kernel fusing techniques we proposed in Section 3.4, evaluated for $|N| \in \{14, 15, 16\}$.

Our first optimization involves the fusion of `cudaMemcpy` operations. By reorganizing the data sequence, we simplify the data copying process, enabling it to be executed with a single `cudaMemcpy` call instead of multiple calls. This change nearly doubles the efficiency of the process. In the second optimization, we fuse the correction step into the `ModDown` operation in the BGV scheme. This integration allows the addition of the correction term directly into the residue-wise arithmetic operation in `ModDown`, effectively utilizing RF and reducing the need for extensive GMEM load and store requests. As a result, we observe a speedup ranging from 1.73× to 1.90× across the three parameter sets. The third optimization fuses the scaling operation into the INWT kernels. This approach significantly reduces overhead associated with scaling computations, primarily due to decreased memory access requirements. Here, we achieve a speedup of up to 1.48×. These results demonstrate the substantial effectiveness of our proposed optimizations in enhancing the overall performance of the operations.

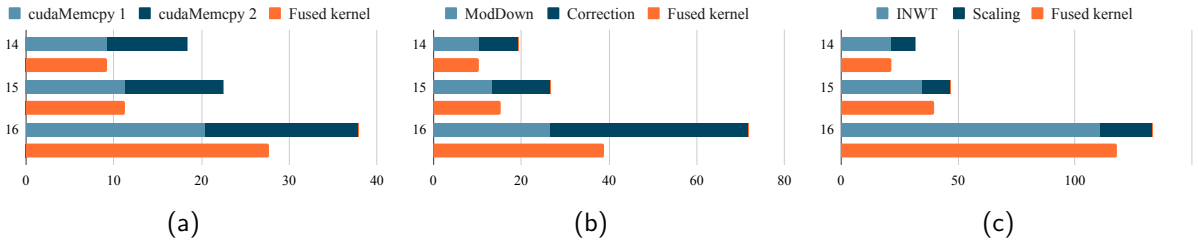

Fig. 7. Execution time comparison before and after kernel fusion (in μs) for $|N| \in \{14, 15, 16\}$: (a) impact of fusing `cudaMemcpy` operations; (b) efficiency gained by integrating `ModDown` with correction in BGV; (c) improvements from combining the NWT with the scaling operation.

Table 6. Performance of our BGV implementation on NVIDIA Tesla A100 GPU.

	Execution time (μs)													
	SEAL	This work												
$\|N\|$	15	12	13	13	13	14	14	14	15	15	15	16	16	16
$\|Q\|$	825	70	162	180	144	378	396	288	825	828	576	1,721	1,729	1,152
$\|QP\|$	881	108	218	218	218	438	438	438	881	881	881	1,777	1,777	1,777
L	14	1	2	4	3	6	10	7	14	22	15	30	46	31
k	1	1	1	1	2	1	1	4	1	1	8	1	1	16
`dnum`	15	2	3	5	2	7	11	2	15	23	2	31	47	2
$\texttt{KeyGen}_{\texttt{sk}}$	7692	27.6	30.7	31.7	31.7	39.9	46.1	45.0	86.0	116.7	116.7	270.3	398.3	398.3
$\texttt{KeyGen}_{\texttt{pk}}$	15444	45.1	50.2	51.2	52.2	61.4	70.6	69.6	126.9	172.0	172.0	396.2	583.6	583.6
$\texttt{KeyGen}_{\texttt{rlk}}$	226471	94.2	151.6	251.9	107.5	419.8	747.5	145.4	1,888.2	3,927.0	356.3	12,272.6	27,482.1	1,201.1
`Ecd`	220	28.7	29.7	29.7	29.7	40.9	40.9	41.0	58.3	58.3	55.3	78.8	76.8	76.8
`Dcd`	254	32.2	43.6	44.0	44.2	48.2	47.4	47.9	62.3	62.3	57.6	83.6	81.6	81.8
`Enc`	36167	205.8	239.6	275.5	260.1	345.0	432.1	385.0	740.3	1,061.8	916.4	2,089.9	3,101.7	2,909.1
`Dec`	6912	25.6	27.6	28.7	28.7	33.8	37.8	34.8	59.4	82.9	60.4	197.6	302.1	205.8
`HAdd`	311	8.2	9.2	9.2	9.2	10.2	11.3	10.2	14.3	19.4	15.3	71.6	102.4	74.7
`CAdd`	5076	33.8	51.2	81.9	66.6	115.7	179.2	132.1	276.5	457.7	299.0	635.9	962.5	652.2
`HMult`	88570	154.6	182.3	224.3	179.2	314.3	448.5	218.1	949.2	1,772.5	473.0	5,197.8	10,975.2	1,892.3
`CMult`	6410	36.9	55.3	86.0	70.7	119.8	184.3	136.2	280.5	469.0	308.2	672.7	1,015.8	684.0
`HRot`	87009	157.7	184.3	226.3	181.2	311.3	445.4	215.0	941.0	1,752.0	460.8	5,109.7	10,852.4	1,807.3

4.4 Performance of HE Schemes

We summarize the performance of our BGV, BFV, and CKKS implementations in Table 6, 7, and 8, respectively. The tables include execution time of SEAL and Phantom on our platform, and the performance of related works.

In general, the most time-consuming modules are the homomorphic operations that require key-switching, and the execution time of the scheme functions is relatively small. We provide the performance of two cases with different `dnum` when $|N| > 12$. Note that choosing a smaller `dnum` leads to a smaller multiplication depth for leveled HE schemes at the same security level.

Similar to BGV, BFV has a plaintext modulus and performs over finite field. However, in the asymmetric encryption `Enc`, since BFV does not need to deal with the correction in `ModDown`, the computational overhead of this function is smaller compared to BGV in which the correction is required. Moreover, according to our experimental results, the HPS-variant performs better compared with the BEHZ-variant in the `Dec` and `HMult` functions. Compared to [8], we offer more functionality as well as larger parameter sets, such as enabling the hybrid key-switching technique.

The CKKS scheme allows input types of rational numbers and, unlike BGV and BFV, it does not require a plaintext modulus. This leads to slower encoding and decoding of the inputs, where the (`I`)`FFT` needs

Table 7. Performance of our BFV implementation and the comparison with [8, 38].

	Execution time (μs)															
	SEAL	[8][1]	[38][2]	This work												
$\|N\|$	15	14	15	12	13	13	13	14	14	14	15	15	15	16	16	16
$\|Q\|$	825	720		70	162	180	144	378	396	288	825	828	576	1,721	1,729	1,152
$\|QP\|$	881		881	108	218	218	218	438	438	438	881	881	881	1,777	1,777	1,777
L	14	12		1	2	4	3	6	10	7	14	22	15	30	46	31
k	1			1	1	1	2	1	1	4	1	1	8	1	1	16
dnum	15			2	3	5	2	7	11	2	15	23	2	31	47	2
$\texttt{KeyGen}_{\texttt{sk}}$	7,692			27.6	30.7	31.7	31.7	39.9	45.0	45.0	86.0	116.7	116.7	270.3	398.3	398.3
$\texttt{KeyGen}_{\texttt{pk}}$	15,444	174,508		41.9	47.1	48.1	48.1	57.3	65.5	65.5	120.8	163.8	165.8	379.9	560.1	560.1
$\texttt{KeyGen}_{\texttt{rlk}}$	226,471			88.0	142.3	235.5	101.3	392.2	699.4	137.2	1,798.1	3,743.7	342.0	11,794.4	26,354.7	1,152.0
Ecd	223			26.6	29.7	29.7	29.7	39.9	40.9	39.9	57.3	57.3	54.2	75.7	75.7	75.7
Dcd	256			32.8	43.1	43.1	43.1	47.3	47.2	47.2	60.2	60.0	56.4	80.6	81.0	78.8
Enc	23,049	3,296		104.4	115.7	119.8	120.8	150.5	174.0	174.0	329.7	451.5	455.6	1,106.9	1,634.3	1,813.5
$\texttt{Dec}_{\text{BEHZ}}$	11,294	252		59.3	61.4	63.5	62.4	70.6	74.7	71.6	120.8	156.6	124.9	349.1	551.9	360.4
$\texttt{Dec}_{\text{HPS}}$				46.0	48.1	49.1	49.1	55.3	59.4	57.3	101.3	134.1	105.4	309.2	483.3	316.4
HAdd	1,078	53	44	8.1	9.2	9.2	9.2	10.2	11.2	10.2	14.3	19.4	14.3	73.7	102.4	74.7
CAdd	1,348			5.1	6.1	6.1	6.1	6.1	7.1	7.1	10.2	12.2	10.2	26.6	38.9	27.6
$\texttt{HMult}_{\text{BEHZ}}$	228,308	11,747	6,907	440.3	492.5	545.8	497.6	695.3	878.6	616.4	1,818.6	3,075.0	1,586.1	8,611.8	17,795.1	6,797.3
$\texttt{HMult}_{\text{HPS}}$				281.6	323.5	387.0	334.8	562.1	766.9	489.4	1,732.6	3,057.6	1,332.2	9,101.3	20,055.0	6067.2
CMult	21,071			82.9	88.0	89.1	88.0	97.2	101.3	99.3	182.2	250.8	189.4	555.0	818.1	567.3
HRot	86,788		3464.5	100.3	126.9	168.9	129.0	247.8	379.9	158.7	863.2	1,674.2	379.9	4,981.7	10,759.2	1,610.7

[1]The performance of [8] is obtained on an NVIDIA Tesla P100 GPU.
[2]The performance of [38] is obtained on an NVIDIA RTX 3060 Ti GPU.

to be computed. Additionally, since CKKS does not need to raise the modulus of plaintexts to add with ciphertexts in the CAdd function, the overall computational overhead difference of its homomorphic addition is not significant. Keeping the ciphertexts in the double-CRT representation makes CKKS has lower complexity in the implementation of operations such as homomorphic multiplication, which shows an advantage in speed compared to BFV.

5 Conclusion

In this work, we propose optimized GPU implementations of BGV, BFV and CKKS, and evaluate the performance. We reduce the computational and memory overhead of operations and show methods to achieve optimal performance under parameters of different magnitudes. We develop a framework to integrate the implementation of the three schemes, and provide a thorough benchmark of the implemented schemes. For the deployment of HE schemes in privacy-preserving applications, our experimental results provide a reference for scheme selection and implementation.

References

1. FullRNS-HEAAN. https://github.com/KyoohyungHan/FullRNS-HEAAN (Jan 2022)
2. HEAAN. https://github.com/snucrypto/HEAAN (Jan 2023)
3. HElib. https://github.com/homenc/HElib (Jan 2023)
4. OpenFHE. https://github.com/openfheorg/openfhe-development (Jan 2023)
5. PALISADE. https://gitlab.com/palisade (Jan 2023)
6. Al Badawi, A., Polyakov, Y., Aung, K.M.M., Veeravalli, B., Rohloff, K.: Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme. IEEE Transactions on Emerging Topics in Computing **9**(2), 941–956 (2021). https://doi.org/10.1109/tetc.2019.2902799
7. Al Badawi, A., Veeravalli, B., Lin, J., Xiao, N., Kazuaki, M., Khin Mi Mi, A.: Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters. IEEE Transactions on Parallel and Distributed Systems **32**(2), 379–391 (2021). https://doi.org/10.1109/tpds.2020.3021238

Table 8. Performance of our CKKS implementation and the comparison with [31].

	Execution time (μs)															
	SEAL	[31][1]	This work													
$\|N\|$	15	16	13	13	14	14	15	15	15	15	16	16	16	16	16	16
$\|Q\|$	825	1,693	100	140	340	260	820	740	620	500	1,700	1,620	1,580	1,460	1,420	1,220
$\|QP\|$	881	2,364	160	200	400	380	880	860	800	740	1,760	1,740	1,760	1,700	1,720	1,580
L	14	32	1	2	7	5	19	17	14	11	41	39	38	35	34	29
k	1	11	1	1	1	2	1	2	3	4	1	2	3	4	5	6
`dnum`	15	3	2	3	8	3	20	9	5	3	42	20	13	9	7	5
$\texttt{KeyGen}_{\texttt{sk}}$	7692		29.7	30.7	40.9	39.9	105.4	101.3	94.2	86.0	358.4	350.2	350.2	334.8	334.8	303.1
$\texttt{KeyGen}_{\texttt{pk}}$	15444		44.0	47.1	59.4	58.3	149.5	142.3	131.0	120.8	502.7	491.5	491.5	469.0	470.0	424.9
$\texttt{KeyGen}_{\texttt{rlk}}$	226471		93.1	141.3	461.8	175.1	2,973.7	1,280.0	663.5	370.6	21,086.2	9,837.5	6,408.2	4,259.8	3,318.7	2,155.5
`Ecd`	6318		95.2	97.2	123.9	131.0	224.2	236.5	222.2	213.0	449.5	444.4	436.2	432.1	475.1	460.8
`Dcd`	30055		79.6	82.0	148.3	118.7	943.3	799.9	563.8	354.6	17,726.7	14,386.5	14,635.9	10,875.2	11,152.4	6,910.4
`Enc`	29874		166.9	172.0	215.0	209.9	507.9	491.5	452.6	407.5	1,668.1	1,631.2	1,633.2	1,562.6	1,558.5	1,411.0
`Dec`	926		9.2	8.2	11.2	10.2	19.4	18.4	17.4	15.3	93.1	89.0	87.0	80.9	77.8	65.5
`HAdd`	944	162	8.2	9.2	10.2	10.2	17.4	16.3	14.3	13.3	93.1	89.0	88.0	82.9	79.8	70.6
`CAdd`	430		6.1	6.1	6.1	6.1	9.2	9.2	8.2	8.2	48.1	46.0	45.0	41.9	41.9	36.8
`HMult`	87012	2,960	144.3	164.8	330.7	203.7	1,395.7	816.1	500.7	356.3	8,564.7	5,088.2	3,620.8	2,601.9	2,188.3	1,608.7
`CMult`	1426	135	9.2	9.2	11.2	10.2	21.5	20.4	18.4	16.3	97.28	93.1	91.1	86.0	83.9	72.7
`Rescale`	9270	490	79.8	80.9	93.1	89.0	164.8	157.7	141.3	125.9	500.7	478.2	467.9	436.2	422.9	370.6
`HRot`	85159	2,550	147.4	166.9	328.7	202.7	1,381.3	803.8	491.5	348.1	8,492.0	4,968.4	3,530.7	2,503.6	2,088.9	1,529.8

[1]The performance of [31] is obtained on an NVIDIA Tesla V100 GPU.

8. Al Badawi, A., Veeravalli, B., Mun, C.F., Aung, K.M.M.: High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(2), 70–95 (2018). https://doi.org/10.13154/tches.v2018.i2.70-95
9. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9**(3), 169–203 (2015). https://doi.org/10.1515/jmc-2015-0016
10. Badawi, A.A., Hoang, L., Mun, C.F., Laine, K., Aung, K.M.M.: Privft: Private and fast text classification with homomorphic encryption. IEEE Access **8**, 226544–226556 (2020). https://doi.org/10.1109/access.2020.3045465
11. Bajard, J., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: Selected Areas in Cryptography - SAC 2016. pp. 423–442. Lecture Notes in Computer Science (2017). https://doi.org/10.1007/978-3-319-69453-5_23
12. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Advances in Cryptology - CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 868–886 (2012). https://doi.org/10.1007/978-3-642-32009-5_50
13. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Innovations in Theoretical Computer Science 2012. pp. 309–325 (2012). https://doi.org/10.1145/2090236.2090262
14. Brutzkus, A., Gilad-Bachrach, R., Elisha, O.: Low latency privacy preserving inference. pp. 812–821 (2019)
15. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1243–1255 (2017). https://doi.org/10.1145/3133956.3134061, `https://doi.org/10.1145/3133956.3134061`
16. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. In: Selected Areas in Cryptography - SAC 2018. pp. 347–368. Lecture Notes in Computer Science (2019). https://doi.org/10.1007/978-3-030-10970-7_16
17. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology - ASIACRYPT 2017. Lecture Notes in Computer Science, vol. 10624, pp. 409–437 (2017). https://doi.org/10.1007/978-3-319-70694-8_15
18. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: Fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020). https://doi.org/10.1007/s00145-019-09319-x
19. Crandall, R., Fagin, B.: Discrete weighted transforms and large-integer arithmetic. Mathematics of computation **62**(205), 305–324 (1994)
20. Dowlin, N., Gilad-Bachrach, R., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016. JMLR Workshop and Conference Proceedings, vol. 48, pp. 201–210 (2016). https://doi.org/10.5555/3045390.3045413

21. Ducas, L., Micciancio, D.: FHEW: bootstrapping homomorphic encryption in less than a second. In: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. pp. 617–640 (2015). https://doi.org/10.1007/978-3-662-46800-5_24, https://doi.org/10.1007/978-3-662-46800-5_24
22. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012)
23. Fan, S., Wang, Z., Xu, W., Hou, R., Meng, D., Zhang, M.: Tensorfhe: Achieving practical computation on encrypted data using gpgpu. arXiv preprint arXiv:2212.14191 (2022)
24. Gentleman, W.M., Sande, G.: Fast fourier transforms: for fun and profit. In: American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA. AFIPS Conference Proceedings, vol. 29, pp. 563–578. AFIPS / ACM / Spartan Books, Washington D.C. (1966). https://doi.org/10.1145/1464291.1464352, https://doi.org/10.1145/1464291.1464352
25. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University (2009)
26. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009 (2009). https://doi.org/10.1145/1536414.1536440
27. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: Advances in Cryptology - CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 850–867 (2012). https://doi.org/10.1007/978-3-642-32009-5_49
28. Halevi, S., Polyakov, Y., Shoup, V.: An improved rns variant of the bfv homomorphic encryption scheme. In: Topics in Cryptology - CT-RSA 2019. Lecture Notes in Computer Science, vol. 11405, pp. 83–105 (2019). https://doi.org/10.1007/978-3-030-12612-4_5
29. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Topics in Cryptology - CT-RSA 2020. pp. 364–390. Lecture Notes in Computer Science (2020). https://doi.org/10.1007/978-3-030-40186-3_16
30. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (2014). https://doi.org/10.1016/j.jsc.2013.09.002
31. Jung, W., Kim, S., Ahn, J.H., Cheon, J.H., Lee, Y.: Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(4), 114–148 (2021). https://doi.org/10.46586/tches.v2021.i4.114-148
32. Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk. vol. 145, pp. 293–294 (1962)
33. Kim, A., Polyakov, Y., Zucca, V.: Revisiting homomorphic encryption schemes for finite fields. In: Advances in Cryptology - ASIACRYPT 2021. pp. 608–639. Lecture Notes in Computer Science (2021). https://doi.org/10.1007/978-3-030-92078-4_21
34. Kim, S., Jung, W., Park, J., Ahn, J.H.: Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In: IEEE International Symposium on Workload Characterization, IISWC 2020. pp. 264–275 (2020). https://doi.org/10.1109/IISWC50251.2020.00033
35. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
36. Lu, W.j., Zhou, J.j., Sakuma, J.: Non-interactive and output expressive private comparison from homomorphic encryption. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018. pp. 67–74 (2018). https://doi.org/10.1145/3196494.3196503
37. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. Journal of the ACM **60**(6), 1–35 (2013). https://doi.org/10.1145/2535925
38. Özcan, A.S., Ayduman, C., Türkoglu, E.R., Savas, E.: Homomorphic encryption on GPU. IACR Cryptol. ePrint Arch. p. 1222 (2022), https://eprint.iacr.org/2022/1222
39. Özerk, Ö., Elgezen, C., Mert, A.C., Öztürk, E., Savas, E.: Efficient number theoretic transform implementation on GPU for homomorphic encryption. J. Supercomput. **78**(2), 2840–2872 (2022). https://doi.org/10.1007/S11227-021-03980-5, https://doi.org/10.1007/s11227-021-03980-5
40. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: Imagenet large scale visual recognition challenge. International Journal of Computer Vision **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
41. Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL (Jan 2023), microsoft Research, Redmond, WA.
42. Shen, S., Yang, H., Liu, Y., Liu, Z., Zhao, Y.: CARM: CUDA-accelerated RNS multiplication in word-wise homomorphic encryption schemes for internet of things. IEEE Transactions on Computers (2022)