Implementing and Benchmarking Word-Wise Homomorphic Encryption Schemes on GPU

Hao Yang[1], Shiyu Shen[2,4], Wangchen Dai[3], Lu Zhou[1], Zhe Liu[3,1] and Yunlei Zhao[2,4]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China, crypto@d4rk.dev

[2] School of Computer Science, Fudan University, Shanghai, China, shenshiyu21@m.fudan.edu.cn

[3] Zhejiang Lab, Hangzhou, China, w.dai@my.cityu.edu.hk

[4] State Key Laboratory of Cryptology, Beijing, China

Abstract. Homomorphic encryption (HE) is one of the most promising techniques for privacy-preserving computations, especially for word-wise HE schemes that allow batched computations over ciphertexts. However, the high computational overhead hinders the deployment of HE in real-word applications. The GPUs are often used to accelerate the execution in such scenarios, while the performance of different HE schemes on the same GPU platform is still absent.

In this work, we implement three word-wise HE schemes BGV, BFV, and CKKS on GPU, with both theoretical and engineering optimizations. We optimize the hybrid key-switching technique, reducing the computational and memory overhead of this procedure. We explore several kernel fusing strategies to reuse data, which reduce the memory access and IO latency, and improve the overall performance. By comparing with the state-of-the-art works, we demonstrate the effectiveness of our implementation.

Meanwhile, we present a 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.

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

1 Introduction

Homomorphic encryption (HE) is a class of cryptosystem that offers the ability to perform computations over encrypted data without the knowledge of secret keys. This allows the construction of non-interactive and secure computational models that do not require the users to stay online during the entire evaluation process, which inherently have lower communication overhead compared to alternatives based on techniques such as multi-party computation. Currently, homomorphic encryption is considered as a promising building block for privacy-preserving applications, such as secure neural network inference [DGBL+16, BGBE19], private set union and intersection [CLR17], private decision tree evaluation [LZS18], etc.

In 2009, Gentry proposed the concept of bootstrapping [Gen09b, Gen09a], bringing the homomorphic encryption scheme from Somewhat Homomorphic Encryption (SHE), which requires the circuit depth to be predetermined, 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 [LPR13]. According to the type of data and basic operations, these schemes can be classified as bit-wise FHE and word-wise FHE. The first class, such as FHEW [DM15] and TFHE [CGGI20], encrypts a few bits per ciphertext and performs logical operations. Although they feature fast bootstrapping and support computation of both polynomial and non-polynomial functions, the ciphertext-to-message size expansion ratio is huge as they encrypt a single or few bits per ciphertext. 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 [BGV12], BFV [Bra12, FV12] and CKKS [CKKS17], where BGV and BFV perform exact operations on finite fields and CKKS supports approximate computations over real and complex numbers. Compared to bit-wise HE schemes, word-wise HE schemes are more efficient as they support batch processing, where multiple plaintexts can be packed into a ciphertext and evaluated in a single-instruction-multiple-data (SIMD) manner.

Currently, many software HE libraries implement the RNS variants of these schemes [GHS12, BEHZ17, HPS19, CHK+19, HK20], such as HElib [HEl23], PALISADE [PAL23], SEAL [SEA23], HEEAN [hea23, ful22] and OpenFHE [ope23]. Despite the attention HE attracts, its performance is still unable to meet the practical requirements. For example, the low-latency privacy-preserving inference proposed by Brutzkus et al. [BGBE19] reduces the execution time of a single prediction to 2.2 seconds, but it still takes more than 6 hours to finish the entire computations on the MNIST dataset [LBBH98]. However, this dataset is small and only provides 10 classes, which is far from satisfactory since current applications require over 100 times more classes, such as ImageNet (1000 classes) [RDS+15].

Hardware acceleration with methods such as Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs), which enables parallel execution, has often been considered as a solution for alleviating the aforementioned problems. Currently, GPU acceleration plays an important role in areas such as machine learning and has been applied to accelerate homomorphic encryption schemes and privacy-preserving applications in some prior researches [ABVMA18, BHM+20, JKA+21, ABPA+21, ABVL+21, SYL+22]. 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. For one scheme, many of the latest proposed techniques have not yet been introduced into this field. The parameter sizes supported by the existing works are inadequate and some implemented functions can be further optimized. Additionally, focusing on one scheme may be insufficient for real-world applications, as in some cases floating-point numbers are taken as input and approximate computation is required, while precision loss may not be tolerable in some others. 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.

Our Contributions. The main goal of this work is to present latest implementations and thorough benchmarks of the word-wise HE schemes on GPU, covering the RNS variants of BGV [GHS12], BFV [BEHZ17, HPS19], and CKKS [HK20]. The contributions of our work are summarized below.

- For all three schemes, we provide complete GPU implementations, covering almost all of the scheme functions and homomorphic operations. We optimize the computational and memory overhead of operations such as key switching. Additionally, by using techniques including kernel fusing, we reduce the IO latency of each operation and improve the overall performance.

Table 1: Summary of supported features in related works [ABVMA18, ABPA+21, BHM+20, JKA+21] and this work. The notations are given in Sec 2.1.

Supported features		[ABVMA18]	[ABPA+21]	[BHM+20]	[JKA+21]	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`				✓	✓

- We develop a framework and integrate our GPU implementation of the three schemes. This framework contains comprehensive pre-computation, RNS base management, and memory pool mechanism to provide efficient data access and transfer. We explore several reusing strategies to close the gap between the implementations of the three schemes.

- Based on the developed framework, we provide benchmarks for all functions of the three schemes under different parameter sets. This is the first work that completely reports the performance of BGV on GPU, and our implementations of BFV and CKKS both outperform the state-of-the-art works [ABVMA18, JKA+21] and support larger parameter sizes.

Comparisons with Related Works. There have been several works on accelerating word-wise HE schemes using GPUs, and the most related works are [ABVMA18, ABPA+21, BHM+20, JKA+21, SYL+22]. The comparisons are summarized in Table 1. In detail, the work [ABVMA18] 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 [ABPA+21] 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 [BHM+20] and [JKA+21] 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 [BHM+20] supports `CAdd` and only [JKA+21] supports `HRot`. The work [SYL+22] 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.

Roadmap. The rest of this paper is organized as follows. In Section 2, we provide some necessary preliminaries of homomorphic encryption schemes and the GPU programming model. Section 3 includes the framework structure and implementation details. The benchmarks and comparisons with other works are presented in Section 4. Finally, we conclude this work in Section 5.

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 $R = \mathbb{Z}[X]/(X^N + 1)$ and the corresponding residue ring modulo q as $R_q = R/qR$. 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 $a \overset{\$}{\leftarrow} S$ 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 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 mudulus, and $P = \Pi_{i=0}^{k-1} p_i$ as the special mudulus. During processing, we set the ciphertext mudulus to Q'. For leveled schemes BGV and CKKS, $Q' := Q_\ell = \Pi_{i=0}^{\ell} q_i$ when the ciphertext is at level ℓ, $0 \le \ell \le L$. For scale-invariant scheme BFV, $Q' := Q$. The RNS-decomposition number is $\texttt{dnum} := \lceil (L+1)/k \rceil$. For each ℓ, let $\alpha := k$ and $\beta := \lceil (l+1)/\alpha \rceil$. Below we give the specifications of the implemented schemes.

- 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 $\boldsymbol{a} \overset{\$}{\leftarrow} R_Q$, $\boldsymbol{s} \leftarrow \mathcal{X}$, $\boldsymbol{e} \leftarrow \mathcal{X}_e$, 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 $\mathsf{pk} := (\boldsymbol{b}, \boldsymbol{a}) \in R_Q^2$ and the secret key as $\mathsf{sk} := (1, \boldsymbol{s}) \in R^2$.
  - Key-switching key. Given two secret keys $\mathsf{sk} = (1, \boldsymbol{s})$ and $\mathsf{sk}' = (1, \boldsymbol{s}')$, sample $\boldsymbol{a}'_j \overset{\$}{\leftarrow} R_{PQ}$ 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 $\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}} := \{(\boldsymbol{b}'_j, \boldsymbol{a}'_j)\}_{0 \le j < \texttt{dnum}} \in R_{PQ}^{2 \times \texttt{dnum}}$, where $B_j \in \mathbb{Z}_{Q_L}$ and satisfies that $B_j = 1 \pmod{q_i}$ for $j\alpha \le i < (j+1)\alpha$ and is zero otherwise.
- Encryption. Given a public key $\mathsf{pk} = (\boldsymbol{b}, \boldsymbol{a}) \in R_Q^2$, sample $\boldsymbol{r} \leftarrow \mathcal{X}$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \mathcal{X}_e$, compute $\texttt{Enc}_{\mathsf{pk}}(0) := [\boldsymbol{r} \cdot (\boldsymbol{b}, \boldsymbol{a}) + t'(\boldsymbol{e}_0, \boldsymbol{e}_1)]_Q$. 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, where μ is the correction factor and we set $\mu := 1$. The resulting ciphertext is $\mathsf{ct} := \texttt{Enc}_{\mathsf{pk}}(\boldsymbol{m}) := [\texttt{Enc}_{\mathsf{pk}}(0) + (\boldsymbol{m}^*, 0)]_Q$.

- Decryption. Given a ciphertext $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R_{Q'}^2$ 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 $R_{Q'}^2$, 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 when the correction factors are mismatched before addition.

- Multiplication. Given two ciphertexts $\mathsf{ct}_1 = (\boldsymbol{c}_0^{(1)}, \boldsymbol{c}_1^{(1)}), \mathsf{ct}_2 = (\boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(2)}) \in R_{Q'}^2$, 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 $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in R_{Q'}^2$ 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 $\rho^\varsigma : R \to R$ 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., $R_t = R/tR$ in BGV and BFV and $R = \mathbb{Z}[X]/(X^N + 1)$ 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 $R_t = \mathbb{Z}_t/(X^N + 1)$ 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 R_t. Formally, the decoding in BGV and BFV is given by $\mathtt{BatchDcd} : R_t \to \mathbb{Z}_t^N, p(X) \mapsto (p(\zeta), p(\zeta^3), \ldots, p(\zeta^{2N-1}))$ 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., $\mathtt{CKKSEcd}(z \in \mathbb{C}^{N/2}; \Delta) : \mathbb{C}^{N/2} \to R, z \mapsto m = \lfloor \Delta \cdot \sigma^{-1}(\pi^{-1}(z)) \rceil_R$ and $\mathtt{CKKSDcd}(m \in R; \Delta) : R \to \mathbb{C}^{N/2}, m \mapsto z = \pi \circ \sigma(\Delta^{-1} \cdot m)$, 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 $\boldsymbol{f} \in R_{Q_\ell}$ can be represented as $(\boldsymbol{f}_0, \ldots, \boldsymbol{f}_\ell) \in \Pi_{i=0}^{\ell} R_{q_i}$, and multi-precision arithmetic can be replaced by a set of residues-wise arithmetic through the isomorphism $R_{Q_\ell} \to R_{q_0} \times R_{q_1} \times \cdots \times R_{q_\ell}$. The RNS-friendly feature of BGV and CKKS allows double-CRT representation of ciphertexts most of the time, i.e., $(\hat{\boldsymbol{f}}_0, \ldots, \hat{\boldsymbol{f}}_\ell) \in \Pi_{i=0}^{\ell} R_{q_i}$ 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 $\mathcal{R} = \{r_0, r_1, \ldots, r_{l-1}\}$, i.e., the BEHZ-type [BEHZ17] and the HPS-type [HPS19], which are illustrated as:

- BEHZ: $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}_\ell \to \mathcal{R}}(x) = \left(\left[\sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* \right]_{r_j} \right)_{j=0}^{l-1}$.
- HPS: $\texttt{Conv}^{\text{HPS}}_{\mathcal{Q}_\ell \to \mathcal{R}}(x) = \left(\left[\sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* - \nu Q_\ell \right]_{r_j} \right)_{j=0}^{l-1}$, $\nu = \left\lfloor \sum_{i=0}^{\ell} [x_i \cdot \tilde{q}_i]_{q_i} / q_i \right\rceil$.

Here, $q_i^* = Q_\ell / q_i$ 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 $\texttt{ModUp}_{\mathcal{Q}_\ell \to \mathcal{Q}_\ell \cup \mathcal{R}}([\boldsymbol{c}]_{\mathcal{Q}_\ell}) := ([\boldsymbol{c}]_{\mathcal{Q}_\ell}, \texttt{Conv}_{\mathcal{Q}_\ell \to \mathcal{R}}([\boldsymbol{c}]_{\mathcal{Q}_\ell}))$ and $\texttt{ModDown}_{\mathcal{Q}_\ell \cup \mathcal{R} \to \mathcal{Q}_\ell}([\boldsymbol{c}]_{\mathcal{Q}_\ell}, [\boldsymbol{c}']_{\mathcal{R}}) := ([\boldsymbol{c}]_{\mathcal{Q}_\ell} - \texttt{Conv}_{\mathcal{R} \to \mathcal{Q}_\ell}([\boldsymbol{c}']_{\mathcal{R}})) \cdot [R^{-1}]_{\mathcal{Q}_\ell}$, $R = \Pi_{j=0}^{l-1} r_j$. Here, the term $\delta := R \cdot [[\mathsf{ct}]_R \cdot R^{-1}]_t$ should be added to ensure the correctness for BGV. The $\texttt{Rescale}$ operation scales down the modulus of a given ciphertext $\mathsf{ct} \in R^2_{Q_\ell}$ from Q_ℓ to $Q_{\ell-1}$ by computing $\texttt{ModDown}_{\mathcal{Q}_\ell \to \mathcal{Q}_{\ell-1}}(\mathsf{ct})$.

Key-Switching. We implement an optimized key-switching procedure with the HPS technique [HPS19]. Given a key-switching key $\mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}}$ and a ciphertext $\mathsf{ct}' = (\boldsymbol{c}_0', \boldsymbol{c}_1') \in R^2_{Q'}$ 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 $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = \left[(\boldsymbol{c}_0', 0) + \left\lfloor P^{-1} \cdot [\sum_{j=0}^{\beta-1} c_{1,j}' \cdot \mathsf{ksk}_{\mathsf{sk}' \to \mathsf{sk}, j}]_{PQ'} \right\rceil \right]_{Q'} \in R^2_{Q'}$. 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., $\langle \mathsf{ct}, \mathsf{sk} \rangle = \langle \mathsf{ct}', \mathsf{sk}' \rangle + \boldsymbol{e}_{\texttt{ks}}$, where $\boldsymbol{e}_{\texttt{ks}}$ is the noise.

2.3 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) [CF94]. 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 $\mathbf{d}^{-1} = \{d_i^{-1} \bmod q\}$, the N-point forward and backward NWT of a polynomial $\boldsymbol{f} = \sum_{i=0}^{N-1} f_i X^i$ is formulated as:

$$\hat{\boldsymbol{f}} := \texttt{NWT}_{\mathbf{d},N}(\boldsymbol{f}), \hat{f}_j = \sum_{i=0}^{N-1} f_i d_i \omega^{ij} \pmod{q},$$

$$\boldsymbol{f} := \texttt{INWT}_{\mathbf{d},N}(\hat{\boldsymbol{f}}), f_i = \frac{1}{N \cdot d_i} \sum_{j=0}^{N-1} \hat{f}_j \omega^{-ij} \pmod{q}.$$

Indeed, it turns out that $\texttt{NWT}_{\mathbf{d},N}(\boldsymbol{f}) = \texttt{NTT}_N(\boldsymbol{f} \odot \mathbf{d})$ and $\texttt{INWT}_{\mathbf{d},N}(\hat{\boldsymbol{f}}) = \mathbf{d}^{-1} \odot \texttt{INTT}_N(\hat{\boldsymbol{f}})$ (mod 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 $\mathcal{O}(N \log N)$.

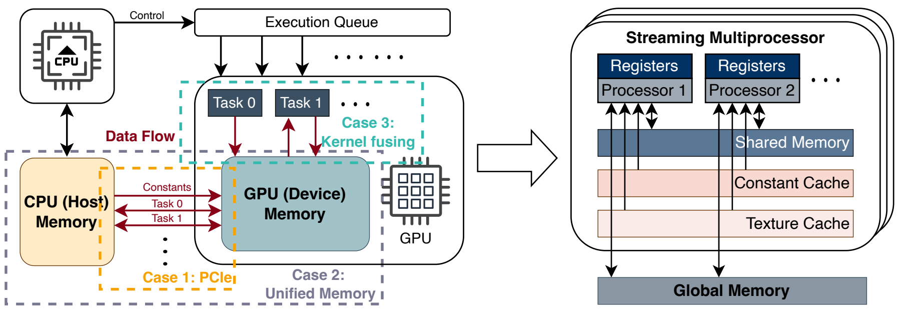


Figure 1: The CPU-GPU computational model and the architecture of GPU. The data communication pattern commonly includes three cases: (1) case 1: transfer data in each task through PCIe; (2) case 2: use unified memory that eliminate host-device data transfer; (3) case 3: fuse kernels and hold data in GPU to omit some transfers.

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 $\{f_i : i = 0, \ldots, n-1\}$ 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.

2.4 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 multiple blocks, and each warp scheduler (WS) in the SM schedules the warps and executes one 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. Another type of platform utilizes unified memory by combining the CPU and GPU memory together to eliminate data transfers. However, the first method may introduce unnecessary data transfers when the tasks are non-independent, while the performance of the second method is barely satisfactory despite the prefetch technique is allowed. 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.

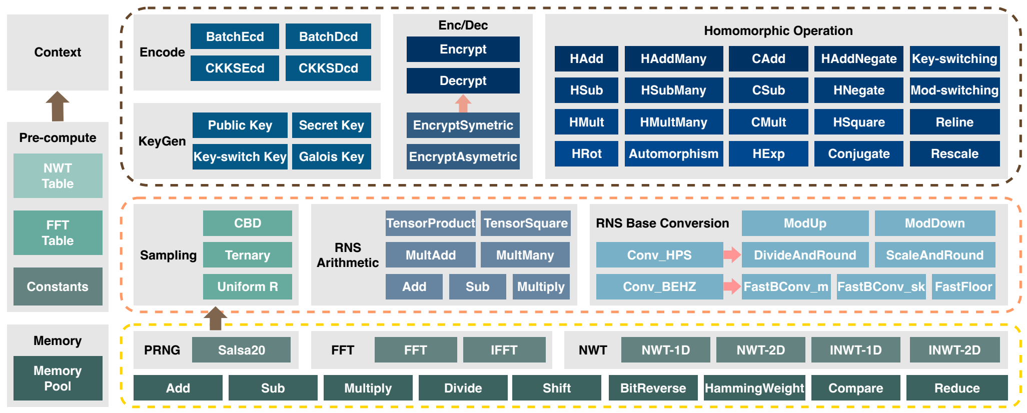


Figure 2: Structure of our benchmarking framework. The implemented functions are organized into three layers, i.e., a math/polynomial layer, an RNS arithmetic layer and a scheme layer.

3 Implementation Details and Optimizations

In this section, we present the implementation details of this work. 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 some 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

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 the optimized implementation of the three 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 implementation written in CUDA PTX assembly to minimize the number of machine instructions and register usage after compiling. 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 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.

3.2 Sampling

The source of the random bytes is obtained from the output of a extendable-output function (XOF). We first generate a 64-byte random seed in the device through the system

API, which is then expanded through the XOF to generate enough pseudo-random bytes for sampling polynomial coefficients. We implement a Salsa20/ChaCha-based [B$^+$08] pseudorandom number generator (PRNG) for the instantiation of the XOF.

The sampling module contains three widely adopted sampling approaches, i.e., a centered binomial distribution sampling, a uniform sampling from ternary distribution R_3 and a uniform sampling from R_{q_i}, $i \in \{0, \ldots, L\}$. The first features generating error polynomials that have negligible statistic difference with Gaussian distributions in an efficient and constant-time way by computing $\sum_{j=0}^{\eta-1}(a_j - b_j)$, which refers to the hamming difference of two η-bit integers a and b. In our implementation, we set $\eta = 21$, which indicates the standard deviation $\sqrt{\eta/2} \approx 3.24$. In the implementation of uniform sampling from R_{q_i}, we first use the PRNG to generate 64 pseudo-random bytes, which are equivalent to eight 64-bit numbers, and then employ the rejection sampling method with a pre-set threshold to obtain elements that meet the requirements of polynomial coefficient size. The ternary sampling that produces elements in $\{-1, 0, 1\}$ uniformly is used in asymmetric encryption. We implement it in a way similar to the uniform sampling from R_{q_i}, with the difference that it does not require rejection.

3.3 NWT

We exploit two approaches to implement NWT, aiming at achieving high parallelism and minimizing the total IO latency in the hierarchical memory of GPU for various magnitudes of input data. In detail, we implement a 1-dimension (I)NWT and 2-dimension (I)NWT that responsible for processing polynomials of $N < 2^{12}$ and $N \geq 2^{12}$, respectively, with each coefficient fits in the machine word size. The batching mode is also enabled so that multiple polynomials can be transformed simultaneously. For the butterfly operation, we follow the CUDA PTX assembly implementation in [SYL$^+$22], which apply the algorithm proposed by David Harvey [Har14] and the Shoup algorithm [Sho01] for fast modular reduction. We illustrate the implementation details below.

The first classical `NWT-1D` approach forms one kernel and is designed for polynomials with small dimension. The main consideration is to exert the maximum parallelism in a block and at the same time reduce the data access of the GMEM. For both `NWT-1D` and `INWT-1D` kernels, we instantiate them with 1024 threads, which is the maximum number of threads that a block can hold. Each thread loads 2 coefficients to the registers and performs a radix-2 butterfly operation on the residues. Except for the first and last layers, data transfer in each layer is performed only between the RF and the SMEM, which offers relatively low data access latency. This method balances the parallelism and number of launched kernels well, and calling it recursively to process polynomials with $N \geq 2^{12}$ is straightforward. However, recursive calling is not generic enough and require GMEM access of the entire polynomials between two kernels. With the increase of N, it introduces significant overhead of GMEM access and will lead to performance degradation.

Thus, in the hierarchical `NWT-2D` approach that we used for the cases $N \geq 2^{12}$, we follow the design rationale of [KJPA20, JKA$^+$21, SYL$^+$22] that divide the N-point NWT process into two parts through choosing suitable N_1 and N_2, where $N = N_1N_2$. Let $\psi^{2N} \equiv 1 \bmod q$, the $\mathtt{NWT}_{\mathbf{d},N}$ and $\mathtt{INWT}_{\mathbf{d},N}$ with $\mathbf{d} = \{\psi^i : i = 0, \ldots, N-1\}$ can be formulated as:

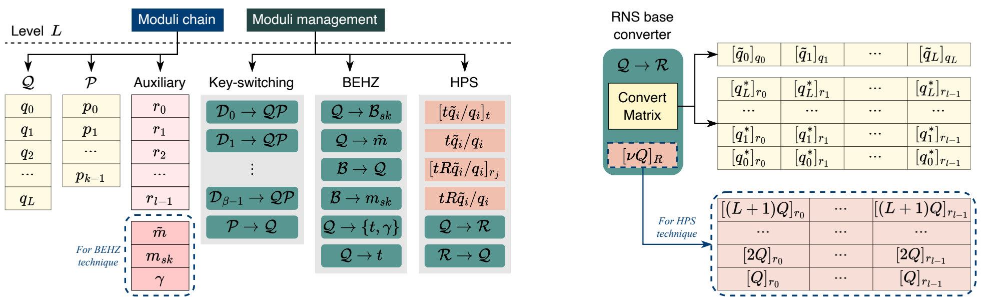


Figure 3: Design of the RNS base management module and base converter in our framework. The auxiliary base and the base converter are initialized depending on the conversion type of BEHZ or HPS.

$$
\begin{aligned}
\hat{x}_{k_1+N_1k_2} &= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2}\psi_N^{2(n_1N_2+n_2)(k_1+N_1k_2)+(n_1N_2+n_2)} \pmod q \\
&= \sum_{n_2=0}^{N_2-1}\left(\psi_N^{2n_2k_1+n_2}\cdot\left(\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2}\psi_{N_1}^{2n_1k_1+n_1}\right)\right)\psi_{N_2}^{2n_2k_2}
\end{aligned}
$$

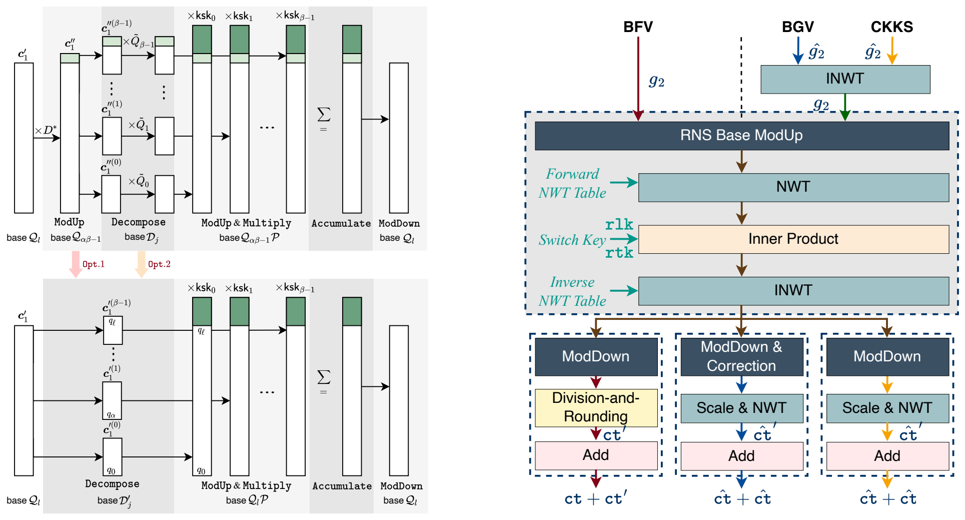
$$
\begin{aligned}
x_{n_1N_2+n_2} &= \frac{1}{N_1N_2}\sum_{k_2=0}^{N_2-1}\sum_{k_1=0}^{N_1-1}\hat{x}_{k_1+N_1k_2}\psi_N^{-2(n_1N_2+n_2)(k_1+N_1k_2)-(n_1N_2+n_2)} \quad (\mathrm{mod} q) \\
&= \frac{1}{N_1}\sum_{k_1=0}^{N_1-1}\left(\psi_N^{-2n_2k_1-n_2}\cdot\frac{1}{N_2}\left(\sum_{k_2=0}^{N_2-1}\hat{x}_{k_1+N_1k_2}\psi_{N_2}^{-2n_2k_2}\right)\right)\psi_{N_1}^{-2n_1k_1-n1}
\end{aligned}
$$

Through this process pattern, we implement two kernels that are responsible for the computation of the two sub-procedures respectively. This approach significantly decreases the number of total GMEM access and reduces IO latency, since we only need to access GMEM between two kernels. In each block, we launch 256 threads and each thread loads 8 coefficients to the RF and performs a radix-8 butterfly operation. Different from previous works, where both [KJPA20] and [JKA$^+$21] only support $|N| \in [14, 17]$, and [SYL$^+$22] supports $|N| \in [11, 17]$ but fixes $N_2 = 2^{11}$, our implementation can handle polynomials of $|N| \leq 17$ and is more flexible.

3.4 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 [BEHZ17] or HPS [HPS19].

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 $\texttt{Conv}^{\mathrm{BEHZ}}_{\mathcal{Q}_\ell\to\mathcal{R}}(\boldsymbol{x})$ and $\texttt{Conv}^{\mathrm{HPS}}_{\mathcal{Q}_\ell\to\mathcal{R}}(\boldsymbol{x})$. 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



(a) Optimized key-switching procedure. (b) Generic design of key-switching module.

Figure 4: The computational flow and generic design of the key-switching module for three schemes. We use the symbol "&" to mark the fused operation in our implementation.

$l \cdot N$ 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.

3.5 Key-Switching

The hybrid key-switching technique is first proposed for CKKS in [HK20] and then introduced into BGV and BFV in [KPZ21]. It is the most practical method, which has been demonstrated in the work [JKA+21] on accelerating CKKS with GPUs. In our implementation, we first utilize two technique 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 [HK20, JKA+21]. 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 $\mathsf{ct}' = (\boldsymbol{c}_0', \boldsymbol{c}_1') \in R_{Q_\ell}^2$ under $\mathsf{sk}' = (1, \boldsymbol{s}')$, this procedure first raise the modulus of $\boldsymbol{c}_1'$ to the nearest $\Pi_{j=0}^{\beta-1} D_j$ by multiplying D^*, such that $\Pi_{j=0}^{\beta-2} D_j < Q_\ell \leq \Pi_{j=0}^{\beta-1} D_j$. 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)NWTs` and point-wise multiplications of polynomials. Additionally, we apply the HPS technique [HPS19] by setting $B_j = \tilde{Q}_j Q_j^*$ in the generation of key-switching keys, $j \in [0, \beta)$, as illustrated in Sec 2.2. 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 4b 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.

Kernel Fusing. Here, we apply three types of kernel fusing to optimize the data access pattern. First, we adapt data copy in `ModUp` to fuse the `Memcpy` operations of base conversion results and digits to reduce IO latency. The `ModUp` operation concatenates the original digit with the RNS base conversion result of the digit. To simplify the computation, we need to adjust the storing order of the residues to follow the order of the modulus chain. Second, in the `ModDown` operation for BGV, we fuse the addition of the correction term $\delta := P \cdot [[\mathsf{ct}']_P \cdot P^{-1}]_t$ into the residue-wise subtraction of the base conversion results. The main consideration is that they have the same parallelism and the modular addition and subtraction here are performed on the same modulus q_i. Third, we fuse the scaling operation into the `NWT` in schemes BGV and CKKS, as the BFV applies a different method to deal with this operation. In the fused kernels, most of the intermediate values between two processes are held in registers to achieve minimum IO latency, reducing the overall data access to the residues. Each block utilizes 256 threads to ensure sufficient SMEM and register resource allocation.

3.6 Homomorphic Multiplication

The multiplication of two given ciphertexts $\mathsf{ct}_1 = (\boldsymbol{c}_0^{(1)}, \boldsymbol{c}_1^{(1)}), \mathsf{ct}_2 = (\boldsymbol{c}_0^{(2)}, \boldsymbol{c}_1^{(2)}) \in R_{Q'}$ consists of two phases of tensor product and relinearization. Figure 5 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- [BEHZ17] and HPS-variant [HPS19] 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 [KO62] 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.

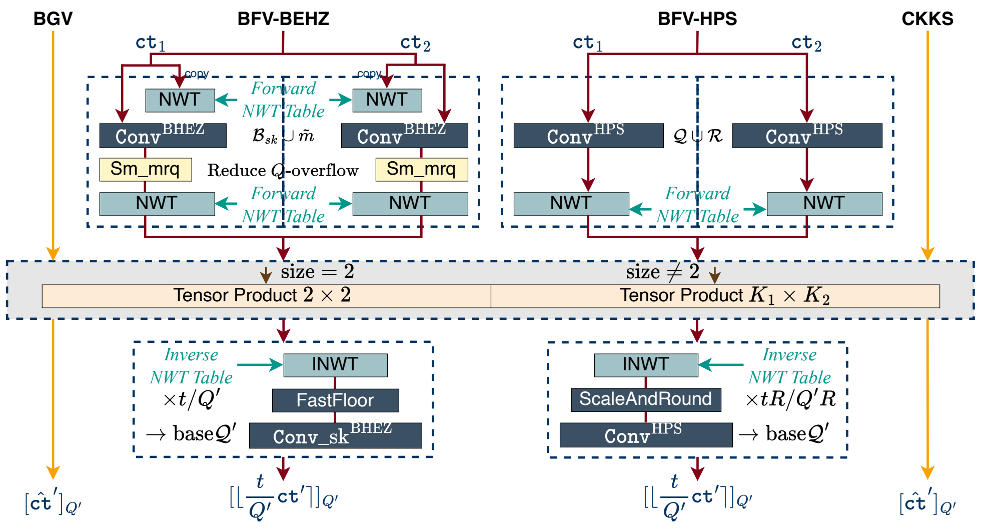


Figure 5: Homomorphic multiplication for three schemes. We apply four branches to make the module compatible with BGV, CKKS, and two variants of BFV.

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 $\mathcal{Q}' \to \mathcal{R}$ and $\mathcal{R} \to \mathcal{Q}'$. For base conversion $\mathcal{Q} \to \mathcal{R}$, 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. We compile the implementations using g++ 12.2.0 and CUDA 11.8 on Arch Linux with kernel 5.15. All experiments are performed on an Intel(R) Core(TM) i9-12900KS CPU with 16 cores and a single NVIDIA GeForce RTX 3090 Ti GPU with 10752 CUDA cores. The performance results are the median of 100 tests. All paramter sets that we used to evaluate the performance of schemes achieve 128 bit security, except in testing the key-switching procedure (in Table 2), for maintaining the same parameter configuration with [JKA+21].

We compare our GPU implementation with the works [ABVMA18, JKA+21], which are the state-of-the-art implementations of BFV and CKKS, respectively. The work [JKA+21] 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 [FWX+22], the authors present a GPU implementation of CKKS using Tensor Core Units. We do not compare our impelemtation with this work because they implement 32-bit arithmetic. The claimed speedup over [JKA+21] comes largely from the data type, as [JKA+21] focuses on 64-bit arithmetic.

Table 2: Performance break down of the hybrid key-switching module of our CKKS implementation and [JKA+21] on RTX 3090 Ti GPU. The functions `ModUp`, `Product`, and `ModDown` denote the modulus-raising, inner product, and modulus-down, respectively.

	Execution time (μs)							Speedup
	[JKA+21]	Our work						
$\lvert N\rvert$	16	16	16	16	16	16	16	
$\lvert Q\rvert$	2260	2260	2260	2260	2260	2260	2260	
$\lvert QP\rvert$	3160	3160	3160	3160	3160	3160	3160	
ℓ	44	44	42	39	36	33	30	
k	15	15	15	15	15	15	15	
`dnum`	3	3	3	3	3	3	3	
`ModUp`	1077[1]	1068	1042	967	912	837	787	
`Product`	367	366	354	338	317	299	280	
`ModDown`	385	384	375	355	338	316	292	
Total	$\geq$ **2214**[2]	2202	2146	2015	1905	1768	**1651**	$>$ **1.3**$\times$

[1]This is obtained from our fixed version because multipling $\tilde{q}_i$ is missing in the released code.
[2]We estimated this result since the open source code and the paper do not provide it.

Performance of arithmetics. We report the performance of the key-switching module and the comparison with [JKA+21] in Table 2. Because the authors do not open source the complete implementation of key-switching, we compare the three steps of it. In their implementation, the `ModUp` and `Product` perform on two ring elements and the `ModDown` performs on one ring element. We follow this logic flow for fair comparison. Besides the three steps, the method used in [JKA+21] requires RNS decomposition before `ModUp`, which introduces no additional overhead in our implementation (see Figure 4a). Nevertheless, we achieved a speedup with the same parameter configuration and moduli chain. As the level of the ciphertext decreases, the execution time of [JKA+21] remains essentially the same because the modulus that the digits raised to is unchanged. In this case, our implementation demonstrates better performance since the total time gradually decreases along with the drop of the modulus, which can speedup the key-switching procedure by more than 1.3×.

Performance of BGV. We summarize the performance of our BGV implementation in Table 3. Since there is little comparable work currently that provides a complete GPU implementation of BGV, we only provide the performance of our implementation obtained on our platform. 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.

Performance of BFV. In Table 4, we list the performance of our BFV-BEHZ and BFV-HPS implementation, and the comparison with [ABVMA18]. Because the authors do not implement `HRot`, the time for generating `rtk` is not included in the total execution time of the `KeyGen` provided in their paper. Compared to their work, we offer more functionality as well as larger parameter sets, such as enabling the hybrid key-switching technique.

Similar to BGV, BFV has a plaintext modulus and performs over finit 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`

Table 3: Performance of our BGV implementation on NVIDIA RTX 3090 Ti GPU.

	Execution time (μs)								
$\lvert N \rvert$	12	13	13	14	14	15	15	16	16
$\lvert Q \rvert$	70	105	142	259	260	500	550	820	980
$\lvert QP \rvert$	107	216	216	436	380	860	850	1,420	1,460
L	1	2	3	5	5	11	14	19	23
k	1	3	2	3	2	6	5	10	8
`dnum`	2	1	2	2	3	2	3	2	3
$\texttt{KeyGen}_{\mathsf{sk}}$	25	30	30	35	36	91	100	300	316
$\texttt{KeyGen}_{\mathsf{pk}}$	38	46	46	54	53	144	159	453	477
$\texttt{KeyGen}_{\mathsf{rlk}}$	80	52	95	115	163	302	489	933	1,470
$\texttt{KeyGen}_{\mathsf{rtk}}$	76	51	89	110	159	310	467	895	1,374
`Ecd`	24	28	28	37	37	52	53	86	86
`Dcd`	32	37	38	43	43	67	67	100	101
`Enc`	185	226	241	309	305	755	853	2,024	2,218
`Dec`	22	24	26	29	29	71	85	225	267
`HAdd`	5	5	5	6	6	26	31	75	88
`CAdd`	32	51	67	96	96	242	302	407	487
`HMult`	150	143	168	194	218	451	566	1,369	1,713
`CMult`	34	53	69	99	99	254	317	443	530
`HRot`	150	142	165	192	216	419	530	1,276	1,603

functions.

Performance of CKKS. We provide our GPU implementation of CKKS and the comparison with [JKA+21] in Table 5. 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 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 three word-wise HE schemes BGV, BFV and CKKS, and evaluate the performance on the same platform. 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. This framework provides sound precomputation, RNS bases management, and memory management. We illustrate reusing and kernel fusing approaches to tune the operations to be compatible with the three schemes on GPU. Finally, we 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.

Table 4: Performance of our BFV implementation and the comparison with [ABVMA18]. The performance of [ABVMA18] is obtained on an NVIDIA Tesla P100 GPU.

	Execution time (μs)									
	[ABVMA18]	Our work								
$\lvert N\rvert$	14	12	13	13	14	14	15	15	16	16
$\lvert Q\rvert$	720	72	105	144	270	270	516	645	900	1,080
$\lvert QP\rvert$		109	216	218	417	368	798	880	1500	1,560
ℓ	12	1	2	3	5	5	11	14	19	23
k		1	3	2	3	2	6	5	10	8
$\mathtt{dnum}$		2	1	2	2	3	2	3	2	3
$\mathtt{KeyGen}_{\mathsf{sk}}$		30	28	28	35	35	93	99	276	294
$\mathtt{KeyGen}_{\mathsf{pk}}$	174,508	43	42	42	51	51	142	151	411	436
$\mathtt{KeyGen}_{\mathsf{rlk}}$		80	48	87	108	156	298	468	858	1,356
$\mathtt{KeyGen}_{\mathsf{rtk}}$		75	49	88	110	158	309	468	893	1,377
$\mathtt{Ecd}$		24	26	26	36	37	50	50	85	85
$\mathtt{Dcd}$		31	36	37	42	44	67	66	101	99
$\mathtt{Enc}$	3,296	90	102	102	144	141	413	461	1,308	1,400
$\mathtt{Dec}_{\mathrm{BEHZ}}$	252	48	48	48	58	58	112	131	362	430
$\mathtt{Dec}_{\mathrm{HPS}}$		39	40	40	51	48	96	110	308	366
$\mathtt{HAdd}$	53	5	5	5	6	6	25	30	75	88
$\mathtt{CAdd}$		6	7	7	12	12	37	45	111	129
$\mathtt{HMult}_{\mathrm{BEHZ}}$	11,747	433	429	460	573	601	1,286	1,641	4,083	5,093
$\mathtt{HMult}_{\mathrm{HPS}}$		175	185	196	325	325	949	1,295	3,398	4,697
$\mathtt{CMult}$		77	80	79	88	88	164	183	493	589
$\mathtt{HRot}$		103	82	111	138	169	359	488	1,125	1,466

Table 5: Performance of our CKKS implementation and the comparison with [JKA+21]. The performance of [JKA+21] is obtained on an NVIDIA Tesla V100 GPU.

	Execution time (μs)									
	[JKA+21]	Our work								
$\lvert N\rvert$	16	12	13	13	14	14	15	15	16	16
$\lvert Q\rvert$	1,693	70	105	142	259	260	500	550	820	980
$\lvert QP\rvert$	2,364	107	216	216	436	380	860	850	1,420	1,460
ℓ	32	1	2	3	5	5	11	14	19	23
k	11	1	3	2	3	2	6	5	10	8
$\mathtt{dnum}$	3	2	1	2	2	3	2	3	2	3
$\mathtt{KeyGen}_{\mathsf{sk}}$		30	28	28	35	35	93	100	278	316
$\mathtt{KeyGen}_{\mathsf{pk}}$		43	42	42	52	51	142	152	411	442
$\mathtt{KeyGen}_{\mathsf{rlk}}$		81	48	87	110	157	298	468	859	1,355
$\mathtt{KeyGen}_{\mathsf{rtk}}$		82	49	88	111	158	309	480	895	1,388
$\mathtt{Ecd}$		97	113	115	138	138	205	209	375	393
$\mathtt{Dcd}$		84	95	97	133	131	442	1,167	10,164	15,995
$\mathtt{Enc}$		141	155	154	196	191	452	486	1373	1,469
$\mathtt{Dec}$		5	5	5	6	6	22	29	75	88
$\mathtt{HAdd}$	162	5	5	5	6	6	24	31	75	89
$\mathtt{CAdd}$		3	3	3	4	4	13	16	38	45
$\mathtt{HMult}$	2,960	139	125	149	178	203	382	484	1,119	1,445
$\mathtt{CMult}$	135	5	5	5	7	7	26	32	75	89
$\mathtt{Rescale}$	490	69	70	72	79	79	141	159	374	437
$\mathtt{HRot}$	2,550	140	124	148	176	201	348	447	1,026	1,329

References

[ABPA+21] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. 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.

[ABVL+21] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters. IEEE Transactions on Parallel and Distributed Systems, 32(2):379–391, 2021.

[ABVMA18] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. 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.

[B+08] Daniel J Bernstein et al. Chacha, a variant of salsa20. In Workshop record of SASC, volume 8, pages 3–5. Lausanne, Switzerland, 2008.

[BEHZ17] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In Selected Areas in Cryptography - SAC 2016, Lecture Notes in Computer Science, pages 423–442, 2017.

[BGBE19] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. pages 812–821, 2019.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Innovations in Theoretical Computer Science 2012, pages 309–325, 2012.

[BHM+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. IEEE Access, 8:226544–226556, 2020.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Advances in Cryptology - CRYPTO 2012, volume 7417 of Lecture Notes in Computer Science, pages 868–886, 2012.

[CF94] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. Mathematics of computation, 62(205):305–324, 1994.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Journal of Cryptology, 33(1):34–91, 2020.

[CHK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In Selected Areas in Cryptography - SAC 2018, Lecture Notes in Computer Science, pages 347–368, 2019.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Advances in Cryptology - ASIACRYPT 2017, volume 10624 of Lecture Notes in Computer Science, pages 409–437, 2017.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. 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, pages 1243–1255, 2017.

[DGBL+16] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, volume 48 of JMLR Workshop and Conference Proceedings, pages 201–210, 2016.

[DM15] Léo Ducas and Daniele Micciancio. 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, pages 617–640, 2015.

[ful22] FullRNS-HEAAN. https://github.com/KyoohyungHan/FullRNS-HEAAN, January 2022.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch., page 144, 2012.

[FWX+22] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. arXiv preprint arXiv:2212.14191, 2022.

[Gen09a] Craig Gentry. A Fully Homomorphic Encryption Scheme. PhD thesis, Stanford University, 2009.

[Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. In Advances in Cryptology - CRYPTO 2012, volume 7417 of Lecture Notes in Computer Science, pages 850–867, 2012.

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation, 60:113–119, 2014.

[hea23] HEAAN. https://github.com/snucrypto/HEAAN, January 2023.

[HEl23] HElib. https://github.com/homenc/HElib, January 2023.

[HK20] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In Topics in Cryptology - CT-RSA 2020, Lecture Notes in Computer Science, pages 364–390, 2020.

[HPS19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In Topics in Cryptology - CT-RSA 2019, volume 11405 of Lecture Notes in Computer Science, pages 83–105, 2019.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 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.

[KJPA20] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In IEEE International Symposium on Workload Characterization, IISWC 2020, pages 264–275, 2020.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In Doklady Akademii Nauk, volume 145, pages 293–294, 1962.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In Advances in Cryptology - ASIACRYPT 2021, Lecture Notes in Computer Science, pages 608–639, 2021.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Journal of the ACM, 60(6):1–35, 2013.

[LZS18] Wen-jie Lu, Jun-jie Zhou, and Jun Sakuma. 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, pages 67–74, 2018.

[ope23] OpenFHE. https://github.com/openfheorg/openfhe-development, January 2023.

[PAL23] PALISADE. https://gitlab.com/palisade, January 2023.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. International Journal of Computer Vision, 115(3):211–252, 2015.

[SEA23] Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[Sho01] Victor Shoup. NTL: A library for doing number theory. https://libntl.org, 2001.

[SYL+22] Shiyu Shen, Hao Yang, Yu Liu, Zhe Liu, and Yunlei Zhao. CARM: CUDA-accelerated RNS multiplication in word-wise homomorphic encryption schemes for internet of things. IEEE Transactions on Computers, 2022.