Leveraging GPU in Homomorphic Encryption: Framework Design and Analysis of BFV Variants

Shiyu Shen, Hao Yang, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao

***Abstract*—Homomorphic Encryption (HE) enhances data security by enabling computations on encrypted data, advancing privacy-focused computations. The BFV scheme, a promising HE scheme, raises considerable performance challenges. Graphics Processing Units (GPUs), with considerable parallel processing abilities, offer an effective solution.**

In this work, we present an in-depth study on accelerating and comparing BFV variants on GPUs, including Bajard-Eynard-Hasan-Zucca (BEHZ), Halevi-Polyakov-Shoup (HPS), and recent variants. We introduce a universal framework for all variants, propose optimized BEHZ implementation, and first support HPS variants with large parameter sets on GPUs. We also optimize low-level arithmetic and high-level operations, minimizing instructions for modular operations, enhancing hardware utilization for base conversion, and implementing efficient reuse strategies and fusion methods to reduce computational and memory consumption.

Leveraging our framework, we offer comprehensive comparative analyses. Performance evaluation shows a 31.9× speedup over OpenFHE running on a multi-threaded CPU and 39.7% and 29.9% improvement for tensoring and relinearization over the state-of-the-art GPU BEHZ implementation. The leveled HPS variant records up to 4× speedup over other variants, positioning it as a highly promising alternative for specific applications.

***Index Terms*—Homomorphic Encryption, BFV, GPU acceleration, parallel processing.**

I. Introduction

THE proliferation of server computing has magnified the scale of server-hosted tasks, yielding remotely obtained results while raising data privacy concerns. Homomorphic Encryption (HE) offers a robust solution, allowing computation on encrypted data accessible only by the secret key holder, thereby protecting data in privacy-sensitive applications.

The pioneering introduction of Fully Homomorphic Encryption (FHE) by Gentry [1], [2] ushered in a range of efficient schemes. One notable branch leverages modular arithmetic over finite fields, encompassing the BGV [3] and BFV [4], [5] schemes which are efficient due to their support for batch processing. The BFV scheme, first proposed in [4] and later adjusted to Ring setting in [5], demonstrates better performance in noise control. Consequently, it has been incorporated into prevalent HE libraries like SEAL [6] and OpenFHE [7] and has found extensive application in a variety of privacy-preserving applications, including private neural inference [8], [9], decision tree evaluation [10], and set intersection [11].

Despite the substantial benefits of the BFV scheme, its performance remains unsatisfactory. Unlike the leveled structure of BGV, BFV uses a scale-invariant design with a constant ciphertext modulus. This results in homomorphic operations on ciphertexts with a large modulus size, leading to inefficiencies due to the data's magnitude. Furthermore, it complicates decryption and multiplication due to the division-and-rounding operation. Since approximate arithmetic is incompatible with the Residue Number System (RNS) representation, this operation is challenging in the RNS variant of BFV.

In light of these challenges, various optimizations have emerged to mitigate these constraints. Currently, two primary adaptations of BFV in RNS stand out: the Bajard-Eynard-Hasan-Zucca (BEHZ) [12] method and the Halevi-Polyakov-Shoup (HPS) [13] method. The first introduces approximation and correction algorithms that work on modular integer arithmetic, while the second uses floating-point arithmetic to streamline evaluation. A notable limitation of the HPS approach is the need for high-precision floating-point arithmetic to support a larger ciphertext modulus. The study in [14] addresses this with a digit decomposition technique and offers optimizations for noise control and performance, including optimized homomorphic multiplication and a leveled approach.

However, the high computational demands of these methods significantly hinder the development of practical HE-based applications. GPU acceleration, investigated by numerous studies, is crucial for enhancing BFV performance. Recently, several efforts have been made to accelerate BFV using GPUs, but limitations persist. Firstly, the research primarily focuses on the BEHZ variant, with less attention given to the HPS variant. Current GPU implementation of HPS supports only 32-bit arithmetic, limiting its applicability to larger precisions. Furthermore, recent optimizations for HPS with potentially superior performance remain unexplored.

Contributions. In this work, we comprehensively analyze all BFV variants, break down the operations, and design a generic GPU framework that accommodates and accelerates all these variants. Our contributions are summarized as follows:

- Our framework optimizes the BEHZ variant and is the first to support HPS with large parameter sets and other HPS variants, surpassing state-of-the-art works in functionality and performance.
- We introduce several optimization techniques to enhance performance. For arithmetic operations, we reduce instructions for modular operations and improve base conversion through unrolling. For scheme operations,

S. Shen and Y. Zhao are with Fudan University, Shanghai, China.
H. Yang and L. Zhou are with Nanjing University of Aeronautics and Astronautics, Nanjing, China.
W. Dai is with the Sun Yat-Sen University, Shenzhen, China.
Z. Liu are with the Zhejiang Lab, Hangzhou, China.
H. Yang and W. Dai are the corresponding authors.
Manuscript received July, 2023.

we propose intra-arithmetic fusion and inner-conversion fusion methods to reuse temporary values, reduce base conversions, and decrease computational consumption.
- We propose new adaptation methods. We are the first to adapt the hybrid key-switching technique to all BFV variants in the GPU domain. For leveled variants, we propose a fusion strategy for tensoring and relinearization to reduce scaling computation and memory transfer.

Our performance evaluation reveals significant improvements. Compared to OpenFHE on a multi-threaded CPU, our implementation on A100 achieves speedups of 14.3× to 31.9×. Against a recent state-of-the-art GPU implementation of BEHZ, we achieve a 39.7% improvement for tensoring and a 29.9% improvement for relinearization. For real-world applications, as the ciphertext level decreases, our GPU implementation of the leveled HPS variant can offer around 4× the speed of other variants, showing potential in specific applications.

Code. Our code is integrated into the Phantom library [15], available at https://github.com/encryptorion-lab/phantom-fhe.

Related Work. Badawi et al. [16] first explored the implementation of the BEHZ variant on GPUs, covering key generation, encryption, decryption, homomorphic addition, and homomorphic multiplication procedures. Further efforts have sought to improve the performance [17]–[21], by optimizing the arithmetic operations bottleneck such as Number Theoretic Transform (NTT) or expanding the range of other homomorphic operations. For the HPS variant, the initial GPU implementation was proposed in [22], but was limited to 32-bit arithmetic due to precision constraints. More scalable implementations later emerged, extending to multi-GPU architectures [23]. Efforts to accelerate privacy-preserving applications based on BFV have also been researched, such as homomorphic convolutional neural networks (HCNN) [24] and gradient boosting inference (XGBoost) [19], [20].

II. Preliminaries

A. Notation

We denote q as an integer and N as a power of 2. Let $\mathbb{Z}$ be the set of integers, with $\mathbb{Z}_q$ representing integers modulo q. We define the polynomial ring as $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ and the residue ring modulo q as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. Bold, italic lowercase letters represent polynomials, e.g. $\boldsymbol{f} := \sum_{i=0}^{N-1} f_i X^i$, and $\boldsymbol{f} \in \mathcal{R}$. $\mathcal{U}_Q$ indicates the uniform distribution over $\mathbb{Z}_Q$. $\mathcal{X}_k$ and $\mathcal{X}_e$ denote two distinct probability distributions over $\mathcal{R}$. The notation $\boldsymbol{f} \leftarrow \mathcal{S}$ means $\boldsymbol{f}$ is sampled according to $\mathcal{S}$. $[a]_q$ stands for a modulo q.

B. Residue Number System

The Residue Number System (RNS) is often used to accelerate multi-precision integer arithmetic. Consider a large integer $Q := \Pi_{i=1}^{L} q_i$ where all q_i are pairwise coprime. The Chinese Remainder Theorem (CRT) provides an isomorphism $\mathbb{Z}_Q \simeq \Pi_{i=1}^{L} \mathbb{Z}_{q_i}$, allowing decomposition of a large integer in $\mathbb{Z}_Q$ into residues in $\mathbb{Z}_{q_i}$, each fitting in machine word size. This extends to rings as $\mathcal{R}_Q \simeq \Pi_{i=1}^{L} \mathcal{R}_{q_i}$. For $\boldsymbol{x} \in \mathcal{R}_Q$ with multi-precision integer coefficients, the RNS representation is $(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(L)})$, where $\boldsymbol{x}^{(i)} := [\boldsymbol{x}]_{q_i}$. This replaces inefficient arithmetic over $\mathcal{R}_Q$ with efficient arithmetic over $\mathcal{R}_{q_i}$, via residue-wise computations using native integer data types. Conversions between the two representations are expressed as:

$$\boldsymbol{x} = \left(\sum_{i=1}^{L} \left[\boldsymbol{x}^{(i)} \cdot \left(\frac{Q}{q_i} \right)^{-1} \right]_{q_i} \cdot \frac{Q}{q_i} \right) - \boldsymbol{v} \cdot Q \tag{1}$$

$$\boldsymbol{x} = \left(\sum_{i=1}^{L} \boldsymbol{x}^{(i)} \cdot \left[\left(\frac{Q}{q_i} \right)^{-1} \right]_{q_i} \cdot \frac{Q}{q_i} \right) - \boldsymbol{v}' \cdot Q \tag{2}$$

These forms differ in their upper bounds. Here, $||\boldsymbol{v}||_\infty$ is limited by L, while $||\boldsymbol{v}'||_\infty$ is capped by $L \cdot \max q_i$.

RNS Base Conversion. Converting the RNS base of an element is crucial during computation. Given large integers $Q := \Pi_{i=1}^{L} q_i$ and $R := \Pi_{k=1}^{K} r_k$ with pairwise coprime moduli, the RNS bases are $\mathcal{Q} := \{q_1, q_2, \ldots, q_L\}$ and $\mathcal{B} := \{r_1, r_2, \ldots, r_K\}$. To convert the base of $\boldsymbol{x} \in \mathcal{R}_Q$ from $\mathcal{Q}$ to $\mathcal{B}$, two methods are proposed. The first is the Bajard-Eynard-Hasan-Zucca (BEHZ) method [12], represented as:

$$\mathtt{Conv}_{\mathcal{Q}\to\mathcal{B}}^{\mathrm{BEHZ}}(\boldsymbol{x}) = \left(\left[\sum_{i=1}^{L} \left[\boldsymbol{x}^{(i)} \cdot \tilde{q}_i \right]_{q_i} \cdot q_i^* \right]_{r_k} \right)_{k=1}^{K} \tag{3}$$

Here, $q_i^* = Q/q_i$ and $\tilde{q}_i = [q_i^{*-1}]_{q_i}$. The output is $[[\boldsymbol{x}]_Q + \boldsymbol{v}Q]_R$ rather than $[[\boldsymbol{x}]_Q]_R$, introducing a Q-overflow error.

The second is the Halevi-Polyakov-Shoup (HPS) method [13], which computes an exact conversion, outputting $[[\boldsymbol{x}]_Q]_P$:

$$\mathtt{Conv}_{\mathcal{Q}\to\mathcal{B}}^{\mathrm{HPS}}(\boldsymbol{x}) = \left(\left[\sum_{i=1}^{L} \left[\boldsymbol{x}^{(i)} \cdot \tilde{q}_i \right]_{q_i} \cdot q_i^* - \boldsymbol{v}Q \right]_{r_k} \right)_{k=1}^{K} \tag{4}$$

Here, $\boldsymbol{v}$ is computed as $\boldsymbol{v} = \lfloor \sum_{i=1}^{L} [\boldsymbol{x}^{(i)} \cdot \tilde{q}_i]_{q_i} / q_i \rceil$. This method first computes $\boldsymbol{y}^{(i)} := [\boldsymbol{x}^{(i)} \cdot \tilde{q}_i]_{q_i}$ using integer arithmetic, then $\boldsymbol{z}^{(i)} := \boldsymbol{y}^{(i)}/q_i$ using floating-point arithmetic, and finally sums up $\boldsymbol{z}^{(i)}$ to get $\boldsymbol{v}$.

C. BFV Scheme

The BFV scheme operates over plaintext space $\mathcal{R}_t$ and ciphertext space $\mathcal{R}_Q$, where t and Q are the plaintext and ciphertext moduli, respectively. For efficiency, we use the RNS-variant BFV, representing Q as the product of coprime integers, i.e., $Q = \Pi_{i=1}^{L} q_i$, with P as the special modulus.

Key Generation. This procedure produces keys for encryption and evaluation based on the desired security level, including the public key $\mathbf{pk}$, secret key $\mathbf{sk}$, and relinearization key $\mathbf{rlk}$. Ring elements $\boldsymbol{s} \leftarrow \mathcal{X}_k$, $\boldsymbol{a} \leftarrow \mathcal{U}_Q$, $\boldsymbol{a}'_j \leftarrow \mathcal{U}_{QP}$, and the noise $\boldsymbol{e}, \boldsymbol{e}'_j \leftarrow \mathcal{X}_e$ are sampled. We denote $\mathtt{dnum}$ as the RNS-decomposition number. The keys are defined as:

- Secret key: $\mathbf{sk} := (1, \boldsymbol{s}) \in \mathcal{R}^2$.
- Public key: $\mathbf{pk} := (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}_Q^2$, with $\boldsymbol{b} := [-\boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}]_Q$.
- Relinearization key: $\mathbf{rlk} := \{(\boldsymbol{b}'_j, \boldsymbol{a}'_j)\}_{0 \le j < \mathtt{dnum}} \in \mathcal{R}_{QP}^{2 \times \mathtt{dnum}}$, where $\boldsymbol{b}'_j := [-\boldsymbol{a}'_j \cdot \boldsymbol{s} + \boldsymbol{e}'_j + PT_j\boldsymbol{s}^2]_{QP}$ and T_j is a decomposition base.

In practice, $\mathbf{sk}$ is typically sampled from the uniform ternary distribution $\{-1, 0, 1\}$, and the discrete Gaussian distribution is often used for sampling the noise [25].

Encryption. Given a plaintext $\boldsymbol{m} \in \mathcal{R}_t$, this procedure generates a ciphertext $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$, encoding $\boldsymbol{m}$ in the most significant digits with a scaling $\Delta := \lfloor Q/t \rfloor$. Symmetric and asymmetric encryption differ by the keys used. Sampling $\boldsymbol{a}' \leftarrow \mathcal{U}_Q$, $\boldsymbol{r} \leftarrow \mathcal{X}_k$ and $\boldsymbol{e}_0, \boldsymbol{e}_1, \leftarrow \mathcal{X}_e$, the two versions are:

- Symmetric: set $\boldsymbol{c}_1 := -\boldsymbol{a}'$ and compute $\boldsymbol{c}_0 := [-\boldsymbol{a}' \cdot \boldsymbol{s} + \boldsymbol{e}_0 + \Delta[\boldsymbol{m}]_t]_Q$.
- Asymmetric: compute $\boldsymbol{c}_0 := [\boldsymbol{r} \cdot \boldsymbol{b} + \boldsymbol{e}_0 + \Delta[\boldsymbol{m}]_t]_Q$ and $\boldsymbol{c}_1 := [\boldsymbol{r} \cdot \boldsymbol{a} + \boldsymbol{e}_1]_Q$.

Decryption. Given a ciphertext $\mathbf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_Q^2$ and a secret key $\mathbf{sk} = (1, \boldsymbol{s})$, the decryption procedure recovers the plaintext by computing $\boldsymbol{m} := [\lfloor t/Q \cdot [\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_Q \rceil]_t$.

Multiplication. Homomorphic multiplication of $\mathbf{ct}_1 = (\boldsymbol{c}_{1,0}, \boldsymbol{c}_{1,1})$ and $\mathbf{ct}_2 = (\boldsymbol{c}_{2,0}, \boldsymbol{c}_{2,1}) \in \mathcal{R}_Q^2$ involves two steps:

- Tensoring. First, takes $\mathtt{ct}_1$ and $\mathtt{ct}_2$ as input and perform tensor product, outputs a triple $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*)$, where
  $\boldsymbol{c}_0^* := [\lfloor \Delta^{-1} \boldsymbol{c}_{1,0} \cdot \boldsymbol{c}_{2,0} \rceil]_Q$
  $\boldsymbol{c}_1^* := [\lfloor \Delta^{-1} (\boldsymbol{c}_{1,0} \cdot \boldsymbol{c}_{2,1} + \boldsymbol{c}_{1,1} \cdot \boldsymbol{c}_{2,0}) \rceil]_Q$
  $\boldsymbol{c}_2^* := [\lfloor \Delta^{-1} \boldsymbol{c}_{1,1} \cdot \boldsymbol{c}_{2,1} \rceil]_Q$
- Relinearization. Converts the triple to a ciphertext with two ring elements, $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1) := ([\boldsymbol{c}_0^* + \boldsymbol{c}_0^{\mathrm{r}}]_Q, [\boldsymbol{c}_1^* + \boldsymbol{c}_1^{\mathrm{r}}]_Q)$, where $(\boldsymbol{c}_0^{\mathrm{r}}, \boldsymbol{c}_1^{\mathrm{r}}) := \lfloor P^{-1} \boldsymbol{c}_2^* \cdot \mathbf{rlk} \rceil$.

Note that relinearization is an optional step but can offer better performance in subsequent computations.

D. GPU Fundamentals

GPUs are powerful tools for high-performance computing due to their parallel processing capabilities. The most basic execution unit is the thread, which runs a kernel in parallel with others to enhance speed. Each thread has private local memory and registers. Threads are organized into thread blocks, where all threads can cooperatively execute tasks and share data via shared memory (SMEM). GPUs feature a complex memory hierarchy, including read-and-write memory such as global memory (GMEM), shared memory, and registers, where GMEM being the largest but having high latency.

GPUs use an instruction set for various operations, including arithmetic, logic, and memory access. A notable feature is that GPUs typically have 32-bit registers, so 64-bit operations are built on 32-bit arithmetic. This design consideration is crucial for optimization. For example, memory access can be more efficient with wider instructions. For 64-bit arithmetic, tuning the decomposition and fusion of operations can reduce the total number of instructions, enhancing performance.

III. BFV Variants

This section analyzes the strengths and challenges of the BFV variants. We also explore recent adaptations, including expanding input size in HPS and reducing computational consumption in tensoring. Finally, we discuss relinearization, focusing on the hybrid key-switching within the BFV scheme.

A. Simple Scaling in Decryption

The BFV decryption procedure first computes $\boldsymbol{x} := [\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_Q$ in $\mathcal{R}_Q$. This is then scaled by t/Q and rounded to compute $\boldsymbol{m} = [\lfloor t/Q \cdot \boldsymbol{x} \rceil]_t$, with each coefficient of $\boldsymbol{x}$ undergoing this process. The BEHZ and HPS methods implement this operation differently. The pseudocode for both methods is shown in Algorithm 1 and 2 respectively.

Algorithm 1 $\mathtt{Dec}_{\mathrm{BEHZ}}$: BEHZ-type BFV Decryption

Input: $\mathbf{ct} \in \mathcal{R}_Q^2$, $\mathbf{sk} \in \mathcal{R}^2$
Output: $[\boldsymbol{m}]_t$
1: $\boldsymbol{x} := \langle \mathbf{ct}, \mathbf{sk} \rangle$
2: **for** $i \in \{t, \gamma\}$ **do** ▷ Scaling-and-rounding
3: $\quad \boldsymbol{f}^{(i)} := [\mathtt{Conv}^{\mathrm{BEHZ}}_{Q \to i}([\gamma t \cdot \boldsymbol{x}]_Q) \times [-Q^{-1}]_i]_i$
4: $\tilde{\boldsymbol{f}}^{(\gamma)} := \boldsymbol{f}^{(\gamma)} \mod \gamma$ ▷ Obtain centered remainder
5: $\boldsymbol{m}^{(t)} := [(\boldsymbol{f}^{(t)} - \tilde{\boldsymbol{f}}^{(\gamma)}) \times [\gamma^{-1}]_t]_t$
6: **return** $\boldsymbol{m}^{(t)}$

Algorithm 2 $\mathtt{Dec}_{\mathrm{HPS}}$: HPS-type in BFV Decryption

Input: $\mathbf{ct} \in \mathcal{R}_Q^2$, $\mathbf{sk} \in \mathcal{R}^2$, $\{(\omega_i, \theta_i)\}_{i \in [0,L)}$
Output: $[\boldsymbol{m}]_t$
1: $\boldsymbol{x} := \langle \mathbf{ct}, \mathbf{sk} \rangle$
2: **for** $j \in [0, N)$ **do** ▷ Scaling-and-rounding
3: $\quad w := 0, v := 0.0$ ▷ $\boldsymbol{x} := \{\boldsymbol{x}^{(i)}\}$
4: $\quad$ **for** $i \in [1, L]$ **do** ▷ $\boldsymbol{x}^{(i)} := \sum_{j=0}^{N-1} x_j^{(i)} X^j$
5: $\quad\quad w := w + [x_j^{(i)} \cdot \omega_i]_{q_i}$ ▷ Integral part
6: $\quad\quad v := v + x_j^{(i)} \cdot \theta_i$ ▷ Fractional part
7: $\quad sum := w + v$
8: $\quad m_j := \lfloor sum - sum/t \rceil$ ▷ Modular reduction
9: **return** $\boldsymbol{m} := \sum_{j=0}^{N-1} m_j X^j$

BEHZ Method. The BEHZ method simplifies computation by replacing rounding with flooring, expressed as follows:

$$\left\lfloor \frac{t}{Q} [\boldsymbol{x}]_Q \right\rfloor = \frac{t[\boldsymbol{x}]_Q - [t \cdot \boldsymbol{x}]_Q}{Q}$$

This provides exact integer division under the RNS representation. During base conversion from $\mathcal{Q}$ to t, the term $t[\boldsymbol{x}]_Q$ vanish modulo t. After conversion, the remaining term $[[t \cdot \boldsymbol{x}]_Q]_t$ can be obtained. An auxiliary modulo γ, coprime to both t and Q, corrects the error produced by base conversion and the replacement of rounding. For $i \in \{t, \gamma\}$, we have

$$\mathtt{Conv}^{\mathrm{BEHZ}}_{\mathcal{Q} \to \{t,\gamma\}}(\boldsymbol{x}) \times [-Q^{-1}]_i = \gamma \left([\boldsymbol{m}]_t + t\boldsymbol{\delta}_0 \right) + \left\lfloor \gamma \frac{\boldsymbol{\delta}_1}{Q} \right\rceil - \boldsymbol{\delta}_2$$

Here, the term $\gamma t \boldsymbol{\delta}_0$ disappears under the moduli t and γ. The error $\boldsymbol{\delta}_2$ is the same under modulo t and γ. Using the centered remainder modulo γ reduces $\gamma([\boldsymbol{m}]_t + t\boldsymbol{\delta}_0)$ modulo t to $\gamma[\boldsymbol{m}]_t$. As such, the decryption result $[\boldsymbol{m}]_t$ can be recovered by multiplying $[\gamma^{-1}]_t$ by $\gamma[\boldsymbol{m}]_t$.

HPS Method. This method focuses on coefficients of ring elements. Let $x \in \mathbb{Z}_Q$ and $m \in \mathbb{Z}_t$ denote the coefficients. The RNS representation of x is $(x^{(1)}, x^{(2)}, \ldots, x^{(L)})$. Applying (2) to decompose x, each coefficient of $\boldsymbol{m}$ is computed as follows:

$$\begin{aligned} m := \left\lfloor \frac{t}{Q} x \right\rceil &= \left\lfloor \left(\sum_{i=1}^{L} x^{(i)} \cdot \tilde{q}_i \cdot \frac{t}{q_i} \right) - \nu \cdot Q \cdot \frac{t}{Q} \right\rceil \\ &= \left[\left\lfloor \sum_{i=1}^{L} x^{(i)} \cdot \left(\tilde{q}_i \cdot \frac{t}{q_i} \right) \right\rceil \right]_t \\ &= \left[\left(\sum_{i=1}^{L} x^{(i)} \cdot \omega_i \right) + \left\lfloor \sum_{i=1}^{L} x^{(i)} \cdot \theta_i \right\rceil \right]_t \end{aligned} \tag{5}$$

Here, $t\tilde{q}_i/q_i := \omega_i + \theta_i$, where $\omega_i \in \mathbb{Z}_t$ and $\theta_i \in [-1/2, 1/2)$ can be pre-computed. We then compute $w := [\sum_{i=1}^{L} x^{(i)}\omega_i]_t$ and $v := \lfloor \sum_{i=1}^{L} x^{(i)}\theta_i \rceil$. The decryption result can finally be obtained by computing $[w+v]_t$.

Enlarge the Input Size in HPS. For the HPS method, we can only store $\tilde{\theta}_i = \theta_i + \epsilon_i$ with κ-bit precision, where the error is $|\epsilon_i| < 2^{-\kappa}$. Consequently, the total error term is $\epsilon = \sum_{i=1}^{L} x_i\epsilon_i$. To ensure correct decryption, $||\epsilon||_\infty < 1/4$ is required. However, the 53-bit precision of IEEE 754 double-precision format limits the HPS method for large moduli. This constraint on the magnitude of $L \cdot q_{\max}$, where $q_{\max} = \max_i q_i$, necessitates high precision floating-point arithmetic for large moduli.

To mitigate this, we use the digit decomposition technique from [14]. We define a base $B \in \mathbb{Z}$ such that $B > 2$, and $d_s = \lceil \log q_{\max} / \log B \rceil$. In this setting, $x^{(i)}$ can be decomposed as $x_i = \sum_{j=0}^{d_s-1} x_j^{(i)} \cdot B^j$. Thus, the result m can be computed as:

$$\begin{aligned} m &= \left[\left(\sum_{i=1}^{L} x^{(i)} \cdot \omega_i \right) + \left\lfloor \sum_{i=1}^{L} x^{(i)} \cdot \theta_i \right\rceil \right]_t \\ &= \left[\left(\sum_{i=1}^{L} \sum_{j=0}^{d_s-1} [x_j^{(i)} \cdot \omega_{i,j}]_t \right) + \left\lfloor \sum_{i=1}^{L} \sum_{j=0}^{d_s-1} x_j^{(i)} \cdot \theta_i \right\rceil \right]_t \end{aligned}$$

Here, $t[\tilde{q}_i \cdot B^j]_{q_i}/q_i := \omega_{i,j} + \theta_{i,j}$, where $\omega_{i,j} \in \mathbb{Z}_t$ and $\theta_{i,j} \in [-1/2, 1/2)$ can be pre-computed. This allows for larger input sizes, making HPS method more flexible with large moduli.

B. Complex Scaling in Tensoring

Given ciphertexts $\mathbf{ct}_1 := (\boldsymbol{c}_{1,0}, \boldsymbol{c}_{1,1})$, $\mathbf{ct}_2 := (\boldsymbol{c}_{2,0}, \boldsymbol{c}_{2,1}) \in \mathcal{R}_Q^2$, the tensoring step in BFV homomorphic multiplication computes the tensor product $\mathbf{ct}_1 \times \mathbf{ct}_2$ to yield $\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1, \boldsymbol{c}'_2)$. It then scales this triple down by t/Q to produce $\mathbf{ct}^* = [\lfloor \frac{t}{Q} \cdot \mathbf{ct}' \rceil]_Q \in \mathcal{R}_Q^3$. In this context, it is crucial to calculate the product without modular reduction before scaling. To ensure correctness, we employ an auxiliary modulus $R := \Pi_{k=1}^{K} r_k > Q$, extending each $\boldsymbol{c}_{i,h}$ to modulo QR for product calculation. This auxiliary base is denoted as $\mathcal{B} = \{r_1, \ldots, r_K\}$. To address the complex scaling involved here, BEHZ [12] and HPS [13] propose two distinct approaches, which we summarize in Algorithm 3 and Algorithm 4.

BEHZ Method. As $\texttt{Conv}_{\mathcal{Q}\to\mathcal{B}}^{\text{BEHZ}}(\boldsymbol{c}_{i,h})$ can cause Q-overflow, symbolized as $\boldsymbol{c}_{i,h} + Q\boldsymbol{u}$, an extra modulus $\tilde{m}$ is introduced to deploy the Small Montgomery Reduction (SMR) to mitigate it. Initially, the ciphertexts are multiplied by $\tilde{m}$ and converted to bases $\mathcal{B}_{sk}$ and $\{\tilde{m}\}$ respectively, where the first conversion yields $\boldsymbol{\varsigma}_{i,h} = [\tilde{m}\boldsymbol{c}_{i,h}]_Q + Q\boldsymbol{u}_{i,h}$. Next, the SMR computes:

$$\boldsymbol{\varsigma}'_{i,h} := \left[\left(\boldsymbol{\varsigma}_{i,h} + Q[-\boldsymbol{c}'^{(\tilde{m})}/Q]_{\tilde{m}} \right) \cdot \tilde{m}^{-1} \right]_i$$

This guarantees that for integers τ and ρ, given that $||\boldsymbol{u}||_\infty < \tau$ and $\tilde{m}\rho \geq 2\tau + 1$, we can deduce $\boldsymbol{\varsigma}'_{i,h} \equiv \boldsymbol{\varsigma}_{i,h} \bmod Q$ and $||\boldsymbol{\varsigma}'_{i,h}||_\infty \leq \frac{Q}{2}(1+\rho)$, thereby mitigating the overflow.

In the rounding step, the γ-correction method is insufficient for exact rounding in complex scaling, so the fast RNS flooring method approximates division of Q from $\boldsymbol{c}$ modulo i as: $[(\boldsymbol{c} - \texttt{Conv}_{\mathcal{Q}\to i}^{\text{BEHZ}}([\boldsymbol{c}]_Q)) \times [Q^{-1}]_i]_i$. Thus for each $\boldsymbol{c}'_h$, $h \in [0,2]$, we have:

$$\left(t \cdot [\boldsymbol{c}'_h]_{\mathcal{B}_{sk}} - \texttt{Conv}_{\mathcal{Q}\to\mathcal{B}_{sk}}^{\text{BEHZ}}(t \cdot [\boldsymbol{c}'_h]_{\mathcal{Q}})\right) \times Q^{-1} = \left\lfloor \frac{t}{Q}\boldsymbol{c}'_h \right\rceil + \boldsymbol{u}'_h$$

where $||\boldsymbol{u}'_h||_\infty \leq L$. Finally, an extra modulus m_{sk} is used to convert back to the original base:

$$\texttt{ConvSK}_{\mathcal{B}_{sk}\to\mathcal{Q}}^{\text{BEHZ}}(\tilde{\boldsymbol{c}}_h) := [\texttt{Conv}_{\mathcal{B}\to\mathcal{Q}}^{\text{BEHZ}}(\tilde{\boldsymbol{c}}_h) - \alpha_{sk}R]_Q$$

Here, $\alpha_{sk} := [(\texttt{Conv}_{\mathcal{B}\to m_{sk}}^{\text{BEHZ}}(\boldsymbol{c}) - [\boldsymbol{c}]_{m_{sk}})R^{-1}]_{m_{sk}}$.

HPS Method. In HPS [13], the scaling process is applied first as in (5) to compute $\tilde{\boldsymbol{c}}_h = [\lfloor \frac{tR}{QR} \cdot \boldsymbol{c}'_h \rceil]_R$, $h \in [0,2]$. This involves scaling over the modulus tR and discarding the RNS component of modulus t, which yields:

$$\begin{aligned} \left[\left\lfloor \frac{t}{Q} \cdot \boldsymbol{c}'_h \right\rceil \right]_{r_k} &= \left[\left\lfloor \sum_{i=1}^{L} \boldsymbol{c}_h'^{(i)} \cdot \frac{t\tilde{Q}_i R}{q_i} + \sum_{k=1}^{K} \boldsymbol{c}_h'^{(k)} \cdot t\tilde{Q}_k r_k^* - tv'R \right\rceil \right]_{r_k} \\ &= \left[\left\lfloor \sum_{i=1}^{L} \boldsymbol{c}_h'^{(i)} \cdot \frac{t\tilde{Q}_i R}{q_i} \right\rceil + \sum_{k=1}^{K} \boldsymbol{c}_h'^{(k)} \cdot t\tilde{Q}_k r_k^* - tv'R \right]_{r_k} \\ &= \left[\left\lfloor \sum_{i=1}^{L} \boldsymbol{c}_h'^{(i)} \cdot \frac{t\tilde{Q}_i R}{q_i} \right\rceil + \boldsymbol{c}_h'^{(k)} \cdot \left[t\tilde{Q}_k r_k^* \right]_{r_k} \right]_{r_k} \end{aligned}$$

Algorithm 3 $\texttt{Tensor}_{\text{BEHZ}}$: BEHZ-type BFV Tensoring

Input: $\mathbf{ct}_1 := (\boldsymbol{c}_{1,0}, \boldsymbol{c}_{1,1})$, $\mathbf{ct}_2 := (\boldsymbol{c}_{2,0}, \boldsymbol{c}_{2,1}) \in \mathcal{R}_Q^2$
Output: $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*) \in \mathcal{R}_Q^3$

for $i \in \{1,2\}$, $h \in \{0,1\}$, $k \in [1,K]$ **do**
  $\boldsymbol{\varsigma}_{i,h} := \texttt{Conv}_{\mathcal{Q}\to\mathcal{B}_{sk}}^{\text{BEHZ}}([\tilde{m}\boldsymbol{c}_{i,h}]_Q)$
  $\boldsymbol{r}_{\tilde{m},i,h} := -\texttt{Conv}_{\mathcal{Q}\to\{\tilde{m}\}}^{\text{BEHZ}}([\tilde{m}\boldsymbol{c}_{i,h}]_Q) \cdot Q^{-1} \mod \tilde{m}$
  $\boldsymbol{\varsigma}'_{i,h} := (\boldsymbol{\varsigma}_{i,h} + Q \cdot \boldsymbol{r}_{\tilde{m},i,h}) \cdot \tilde{m}^{-1}$ ▷ $\texttt{SmMrq}$ in $\mathcal{B}_{sk,\tilde{m}}$
  $\boldsymbol{c}'_{i,h} := \boldsymbol{c}_{i,h} || \boldsymbol{\varsigma}'_{i,h}$, $\mathbf{ct}'_i := (\boldsymbol{c}'_{i,0}, \boldsymbol{c}'_{i,1})$ ▷ Concatenate
$\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1, \boldsymbol{c}'_2) := \mathbf{ct}'_1 \times \mathbf{ct}'_2$ ▷ $\mathbf{ct}'$ in $\mathcal{Q} \cup \mathcal{B}_{sk}$
for $h \in [0,2]$ **do** ▷ Scaling-and-rounding in $\mathcal{B}_{sk}$
  $\tilde{\boldsymbol{c}}_h := \left(t \cdot [\boldsymbol{c}'_h]_{\mathcal{B}_{sk}} - \texttt{Conv}_{\mathcal{Q}\to\mathcal{B}_{sk}}^{\text{BEHZ}}(t \cdot [\boldsymbol{c}'_h]_{\mathcal{Q}})\right) \times Q^{-1}$
for $h \in [0,2]$ **do** ▷ $\texttt{ConvSK}$
  $\alpha_{sk} := [(\texttt{Conv}_{\mathcal{B}\to m_{sk}}^{\text{BEHZ}}([\tilde{\boldsymbol{c}}_h]_{\mathcal{B}}) - [\tilde{\boldsymbol{c}}_h]_{m_{sk}})R^{-1}]_{m_{sk}}$
  $\boldsymbol{c}_h^* := \texttt{Conv}_{\mathcal{B}\to\mathcal{Q}}^{\text{BEHZ}}([\tilde{\boldsymbol{c}}_h]_{\mathcal{B}}) - \alpha_{sk}R$
return $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*)$

Algorithm 4 $\texttt{Tensor}_{\text{HPS}}$: HPS-type BFV Tensoring

Input: $\mathbf{ct}_1 := (\boldsymbol{c}_{1,0}, \boldsymbol{c}_{1,1})$, $\mathbf{ct}_2 := (\boldsymbol{c}_{2,0}, \boldsymbol{c}_{2,1}) \in \mathcal{R}_Q^2$
Output: $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*) \in \mathcal{R}_Q^3$

for $i \in \{1,2\}$, $j \in \{0,1\}$ **do**
  $\boldsymbol{\varsigma}_{i,j} := \texttt{Conv}_{\mathcal{Q}\to\mathcal{B}}^{\text{HPS}}(\boldsymbol{c}_{i,j})$
  $\boldsymbol{c}'_{i,j} := \boldsymbol{c}_{i,j} || \boldsymbol{\varsigma}'_{i,j}$, $\mathbf{ct}'_i := (\boldsymbol{c}'_{i,0}, \boldsymbol{c}'_{i,1})$ ▷ Concatenate
$\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1, \boldsymbol{c}'_2) := \mathbf{ct}'_1 \times \mathbf{ct}'_2$ ▷ $\mathbf{ct}'$ in $\mathcal{Q} \cup \mathcal{B}$
for $h \in [0,2]$ **do** ▷ Scaling-and-rounding
  for $j \in [0,N)$ **do** ▷ $\boldsymbol{c}'_h := \{\boldsymbol{c}_h'^{(i)}, \boldsymbol{c}_h'^{(k)}\}$
    $w := 0, v := 0.0$
    for $i \in [1,L]$ **do** ▷ $\boldsymbol{c}_h'^{(i)} := \sum_{j=0}^{N-1} c_{h,j}'^{(i)} x^j$
      $v := v + c_{h,j}'^{(i)} \cdot \theta_i$
    $v := \lfloor v \rceil$ ▷ Fractional part
    for $k \in [1,K]$ **do**
      for $i \in [1,L]$ **do**
        $w := w + [c_{h,j}'^{(i)} \cdot \omega_{i,k}]_{r_k}$
      $w := w + [c_{h,j}'^{(k)} \cdot \lambda_k]_{r_k}$ ▷ Integral part
      $\tilde{c}_{h,j}^{(k)} := [w+v]_{r_k}$ ▷ $\tilde{\boldsymbol{c}}_h^{(k)} := \sum_{j=0}^{N-1} \tilde{c}_{h,j}^{(k)} x^j$
  $\boldsymbol{c}_h^* := \texttt{Conv}_{\mathcal{B}\to\mathcal{Q}}^{\text{HPS}}(\tilde{\boldsymbol{c}}_h)$ ▷ $\tilde{\boldsymbol{c}}_h := \{\tilde{\boldsymbol{c}}_h^{(k)}\}$
return $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*)$

Here, $\tilde{Q}_i := [(q_i^* R)^{-1}]_{q_i}$, $\tilde{Q}_k := [(Q r_k^*)^{-1}]_{r_k}$, and $r_k^* = R/r_k$. Subsequently, $\tilde{\boldsymbol{c}}_h$ is achieved under base $\mathcal{B}$, then converted to obtain $\boldsymbol{c}_h^* \in \mathcal{R}_Q$. The value for $\frac{t\tilde{Q}_i R}{q_i}$ is $W_i + \theta_i'$, with $W_i \in \mathbb{Z}_R$ representing the integral part and $\theta_i' \in [-\frac{1}{2}, \frac{1}{2})$ the fractional part. Here, we pre-compute and store $\omega_{i,k}' := [W_i]_{r_k}$ for each i in the range $[1, L]$ and k in the range $[1, K]$.

Compact HPS. To ensure accurate tensoring, the auxiliary modulus R should be slightly larger than Q, typically achieved by setting $K = L + 1$. However, a more compact method proposed in [14] suggests switching the modulus of one ciphertext from Q to R. Consequently, after multiplication, the noise magnitude transitions from a multiple of Q^2 to a multiple of QR that diminishes modulo QR. This allows for choosing $R \approx Q$, resulting in a compact parameter configuration that reducing K to L, thereby reducing computational complexity. The distinctions from Algorithm 4 are as follows:

- In line 2, the RNS base of one ciphertext is switched from base $\mathcal{Q}$ to $\mathcal{B}$, and then extended to base $\mathcal{Q} \cup \mathcal{B}$.
- During the scaling-and-rounding process, division by P is performed to directly obtain residues in base $\mathcal{Q}$, instead of dividing by Q and then switching to $\mathcal{Q}$.

This method reduces the number of base conversions by 1. Additionally, the smaller value of K decreases the conversion complexity and reduces the number of (I)NTT by $7(K - L)$.

Leveled HPS. Inspired by the BGV scheme's leveled structure [3], a leveled approach can be applied to BFV multiplication [14] using $Q_l := \Pi_{i=1}^{l} q_i$. This reduce both computational and memory complexity. The process is below:

- Scale the ciphertexts by $\frac{Q_l}{Q}$ to obtain $\widetilde{\mathbf{ct}}_1 := \lfloor \frac{Q_l}{Q} \mathbf{ct}_1 \rceil, \widetilde{\mathbf{ct}}_2 := \lfloor \frac{Q_l}{Q} \mathbf{ct}_2 \rceil \in \mathcal{R}_{Q_l}^2$.
- Extend the RNS base to acquire $\mathbf{ct}_1'$ and $\mathbf{ct}_2' \in \mathcal{R}_{Q_l R_l}^2$, where $Q_l = \Pi_{i=1}^{l} q_i$ and $R_l = \Pi_{k=1}^{l} r_k$.
- Perform the tensor product in $\mathcal{R}_{Q_l R_l}$ to obtain $\mathbf{ct}' := \mathbf{ct}_1' \times \mathbf{ct}_2' := (\boldsymbol{c}_0', \boldsymbol{c}_1', \boldsymbol{c}_2') \in \mathcal{R}_{Q_l R_l}^3$.
- Scale $\mathbf{ct}'$ to modulo Q_l by scaling-and-rounding R_l through computation of $\widetilde{\mathbf{ct}} := \lfloor \frac{t}{P} \cdot \mathbf{ct}' \rceil$ to acquire $\widetilde{\mathbf{ct}} := (\tilde{\boldsymbol{c}}_0, \tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) \in \mathcal{R}_{Q_l}^3$.
- Lastly, scale $\widetilde{\mathbf{ct}} \in \mathcal{R}_{Q_l}^3$ up to the original Q by $\mathbf{ct} := \lfloor \frac{Q}{Q_l} \widetilde{\mathbf{ct}} \rceil$ to get the result $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}_Q^3$.

Compared to the compact HPS variant that reduces the size of r_k by 1, the leveled approach reduces the inner loop from L or K to l, at the cost of more base conversions. However, these conversions are more efficient due to the smaller RNS base of the input and output. Selection criteria for l is provided in [14]. Notably, the final scaling computation of $\mathbf{ct} := \lfloor \frac{Q}{Q_l} \widetilde{\mathbf{ct}} \rceil$ can be simplified to multiplying the discarded component of Q to $\widetilde{\mathbf{ct}}$, as $\frac{Q}{Q_l} = \Pi_{i=l+1}^{L} q_i$. This involves multiplying $[\frac{Q}{Q_l}]_{q_i}$ for $i \in [1, l]$ to each residues $\boldsymbol{c}_h^{(i)}$ modulo q_i and padding the $\boldsymbol{c}_h^{(l+1)}$ to $\boldsymbol{c}_h^{(L)}$ components to zero for $h \in [0, 2]$.

C. Relinearization

Hybrid key-switching, proposed in BGV [26] and later adapted to RNS setting [27], can also be adapted for BFV scheme. Here, we select the special modulus as $P = \Pi_{i=1}^{\ell} p_i$ and define the decomposition number as $\texttt{dnum} := \lceil L/\ell \rceil$.

Algorithm 5 `Relin`: Hybrid BFV relinearization

Input: $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*) \in \mathcal{R}_Q^3$, $\mathbf{rlk} := \{(\mathbf{rlk}_{j,0}, \mathbf{rlk}_{j,1})\}$
Output: $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_Q$

1: $(\boldsymbol{t}_0, \boldsymbol{t}_1) := (0, 0)$
2: **for** $j \in [0, \texttt{dnum})$ **do**
3: $\quad \boldsymbol{\varsigma} := [[\boldsymbol{c}_2^*]_{D_j} \cdot \tilde{Q}_j]_{D_j}$ ▷ Decomposition
4: $\quad \bar{\boldsymbol{c}} := \boldsymbol{\varsigma} || \texttt{Conv}_{\mathcal{D}_j \to \mathcal{Q}_j^* \cup \mathcal{P}}(\boldsymbol{\varsigma})$ ▷ Base extension
5: $\quad (\bar{\boldsymbol{c}}_0^{\mathrm{r}}, \bar{\boldsymbol{c}}_1^{\mathrm{r}}) := (\bar{\boldsymbol{c}} \cdot \mathbf{rlk}_{j,0}, \bar{\boldsymbol{c}} \cdot \mathbf{rlk}_{j,1})$ ▷ Switch key
6: $\quad (\boldsymbol{t}_0, \boldsymbol{t}_1) := ([\boldsymbol{t}_0 + \bar{\boldsymbol{c}}_0^{\mathrm{r}}]_{QP}, [\boldsymbol{t}_1 + \bar{\boldsymbol{c}}_1^{\mathrm{r}}]_{QP})$
7: $(\boldsymbol{c}_0^{\mathrm{r}}, \boldsymbol{c}_1^{\mathrm{r}}) := (\lfloor P^{-1} \cdot \boldsymbol{t}_0 \rceil, \lfloor P^{-1} \cdot \boldsymbol{t}_1 \rceil)$ ▷ Scaling down
8: $(\boldsymbol{c}_0, \boldsymbol{c}_1) := ([\boldsymbol{c}_0^* + \boldsymbol{c}_0^{\mathrm{r}}]_Q, [\boldsymbol{c}_1^* + \boldsymbol{c}_1^{\mathrm{r}}]_Q)$
9: **return** $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$

We denote $D_j := \Pi_{i=0}^{\alpha-1} q_{j\alpha+i}$, and $Q_j^* := Q/D_j$, and $\tilde{Q}_j := [Q_j^{*-1}]_{D_j}$. Then, we present the adapted technique in Algorithm 5. The relinearization key, $\mathbf{rlk}$, consists of tuples $\mathbf{rlk} := \{([-\boldsymbol{a}_j \boldsymbol{s} + \boldsymbol{e}_j + PT_j \boldsymbol{s}^2]_{PQ}, \boldsymbol{a}_j)\}_{j \in [0, \texttt{dnum})}$, wherein the decomposition base, T_j, is set as Q_j^*.

During processing, the decomposition is simplified by extracting the appropriate RNS residues and subsequently multiplying them with $\tilde{Q}_j$. Next, the RNS base is extended from $\mathcal{D}_j$ to $\mathcal{Q} \cup \mathcal{P}$, and the inner-product with $\mathbf{rlk}$ yields the result $(\boldsymbol{t}_0, \boldsymbol{t}_1) \in \mathcal{R}_{QP}^2$. These vectors are then scaled down by a factor of $\frac{1}{P}$ to obtain $(\boldsymbol{c}_0^{\mathrm{r}}, \boldsymbol{c}_1^{\mathrm{r}}) \in \mathcal{R}_Q^2$. This output is added to the original ciphertext to complete the process.

Additionally, we apply [13] to enhance efficiency by setting the decomposition base T_j as $\tilde{Q}_j Q_j^*$. This eliminates the need for scalar multiplication with $\tilde{Q}_j$ in the decomposition stage, streamlining the process.

IV. Framework and Optimized Implementation

In this section, we describe the proposed GPU framework and our optimizations. We begin with a design overview, followed by a detailed implementation of each module and the optimization techniques employed. For simplicity, we denote the compact and leveled variant as CHPS and LHPS, respectively.

A. Design Overview

The computational model and the proposed framework are shown in Fig. 1. The CPU functions as a host, delegating

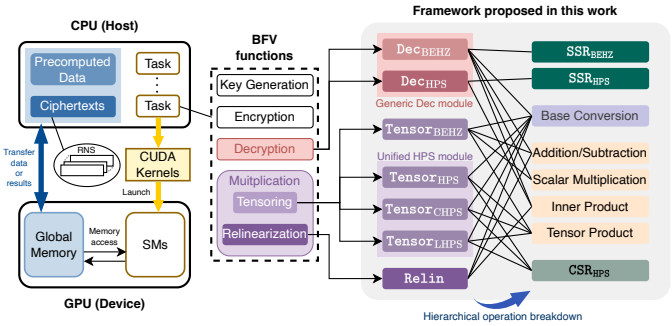

Fig. 1. The computational model and the proposed framework.

Algorithm 6 Modular Multiplication

function ModRed(a, q, μ) ▷ Reduction of 64-bit integer
  $tmp := \_\_umul64hi(a, \mu)$ ▷ $\mu := \lfloor \frac{2^{64}}{q} \rfloor$
  return $a - tmp \cdot q$
function CModMul(a, ζ, q, μ) ▷ Constant multiplication
  $tmp := \_\_umul64hi(a, \mu)$ ▷ $\mu := \lfloor \frac{\zeta \cdot 2^{64}}{q} \rfloor$
  return $a \cdot \zeta - tmp \cdot q$
function ModMul($a, b, q, \mu := \mu_1 \cdot 2^{64} + \mu_0$)
  mul.lo.u64 r_l, a, b
  mul.hi.u64 r_h, a, b ▷ $ab := r_h \cdot 2^{64} + r_l$
  mul.hi.u64 tmp, r_l, μ_0
  mad.lo.cc.u64 tmp, r_l, μ_1, tmp
  madc.hi.u64 $res, r_l, \mu_1, 0$
  mad.lo.cc.u64 tmp, r_h, μ_0, tmp
  madc.hi.u64 res, r_h, μ_0, res
  mad.lo.u64 res, r_h, μ_1, res
  mul.lo.u64 res, res, q ▷ Barrett subtraction
  sub.u64 res, r_l, res
  return res

computational tasks to the GPU by launching corresponding CUDA kernels. Related ciphertexts and pre-computed data are transferred to the GPU, which performs the computations and transfers the results back to the CPU. The tasks involve BFV functions, where in this work we focus on decryption and multiplication. We break down these operations into general sub-modules for better accommodation in the framework.

Hierarchical Operation Breakdown. Based on the earlier analysis, we conduct a hierarchical operation breakdown, deconstructing complex operations into reusable RNS level polynomial operations. As shown in Fig. 1, the HPS variant includes product and specific scaling-and-rounding, while the BEHZ variant requires a more intricate sequence of operations, including product, multiple base conversions, and various arithmetic operations. Consequently, our implementation follows a three-tiered structure supporting diverse computations.

- **Low-level Ring Arithmetics:** This foundational layer sets the groundwork. It begins with our refined integer modular operations. Based on this, we further implement polynomial modular operations to extend the capability.
- **RNS-level Operations:** We perform arithmetic operations over polynomials under the RNS. Leveraging the inherent parallelism of GPU, we batch these operations to enhance performance. These operations are implemented generically for wide applicability and reusability.
- **High-level Module of Homomorphic Operations:** This tier develops generic modules for variant operations, using previously implemented operations. We explore finely-tuned fusion approaches to reduce memory access and usage, thereby boosting the performance.

This layered strategy fosters operational efficiency and versatility, optimizing the execution of both variants and allowing for potential expansion to more computational use cases.

B. Ring Arithmetic

Below, we delve into the backbone implementations of our modular operations and polynomial multiplication.

Modular Reduction. We use Barrett reduction to replace costly division with quicker multiplication and bit-shifting. It necessitates a pre-computated $\mu := \lfloor \frac{2^{\beta}}{q} \rfloor$, with β set as the machine word size. The reduction result is then computed as $a - \lfloor \frac{a \cdot \mu}{2^{\beta}} \rfloor \cdot q$. We develop multiple modular reduction algorithms using CUDA PTX instructions to minimize instruction use, with pseudocode detailed in Algorithm 6. The basic Barrett reduction ModRed reduces a 64-bit integer input a and compute $a \bmod q$. After multiplying a and pre-computed μ, the high 64-bit of the 128-bit multiplication result is stored to implicitly conduct the shifting. This product is then subtracted from q, costing two 64-bit multiplication instructions in total.

Modular Multiplication. We employ the Shoup technique [28] for quicker modular multiplication when one input is constant, denoted as CModMul. When multiplying a and a constant ζ, it can be merged into pre-computation by $\mu := \lfloor \frac{\zeta \cdot 2^{64}}{q} \rfloor$. This saves one multiplication and reducing the cost to that of modular reduction. For cases with two variables, we adopt the optimized modular reduction from [21], developing ModMul for two 64-bit integers. As CUDA does not support 128-bit registers, we use two 64-bit multiplication instructions to obtain the 128-bit result. We calculate the quotient by multiplying each 64-bit component as $\lfloor \frac{(a_1 2^{64} + a_0) \cdot (\mu_1 2^{64} + \mu_0)}{2^{128}} \rfloor$, followed by subtraction from a. The total cost includes 2 multiplication instructions for multiplication and 7 for reduction.

Polynomial Multiplication. This operation poses a significant performance challenge, which we address by employing the NTT. For polynomials $\boldsymbol{f}, \boldsymbol{g} \in \mathcal{R}_q$, the forward negacyclic NTT is formulated as $\hat{f} := \texttt{NTT}(f)$, $\hat{f}_j = \sum_{i=0}^{n-1} f_i \zeta^{(2i+1)j} \pmod q$, and the inverse is $f := \texttt{INTT}(\hat{f})$, $f_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{f}_j \zeta^{-(2i+1)j} \pmod q$, where ζ is the primitive $2n$-th root of unity. This transformation enables multiplication as $fg := \texttt{INTT}(\texttt{NTT}(f) \cdot \texttt{NTT}(g))$, reducing computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. We adopt the commonly used hierarchical NTT implementation [29], [30], and partition the process into two kernels for different transformation levels. For each NTT operation with N elements, we use 8 per-thread implementation, where each thread loads eight residues into the registers and perform a radix-8 transformation. SMEM is used for temporary storage between each radix-8 transformation to reduce GMEM interaction.

C. Base Conversion

Base conversion for a polynomial $\boldsymbol{c}$ in RNS representation is crucial. The sizes of the input and output bases, denoted as I and O, and the process dimensions, $I \cdot N$ and $O \cdot N$, present GPU implementation challenges due to their disparity. Setting the entire parallelism at $I \cdot O \cdot N$, as suggested by [21], could lead to inefficient utilization of GPU hardware resource, while focusing on a single dimension (e.g., $O \cdot N$) burdens thread with repeated calculations. To balance these trade-offs, we implement two distinct kernels with different parallelism.

The first kernel tackles scalar modular multiplication in the $I \cdot N$ dimension, deploying $I \cdot N$ threads to execute the operation, as depicted in Fig. 2b. The subsequent kernel operates in the $O \cdot N$ dimension, executing modular multiplication with a matrix constructed from $[q_i^*]_{q_j}$, as shown in Fig. 2c. For the HPS method, we fuse the fractional computation into the second kernel. Given that $\boldsymbol{v} = \lfloor \sum_{i=1}^{I} [\boldsymbol{x}^{(i)} \cdot \tilde{q}_i]_{q_i} / q_i \rceil$ where

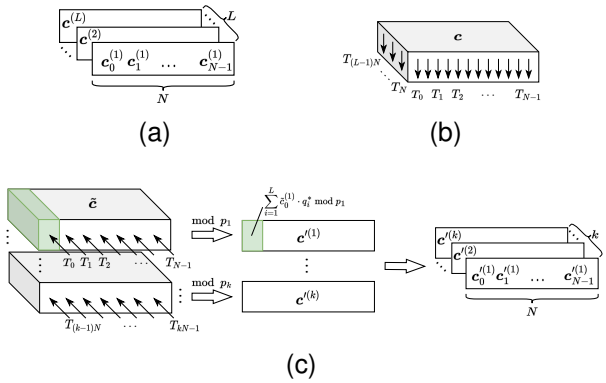

Fig. 2. The structure of our base conversion implementation before loop unroll. (a) A polynomial in RNS representation. (b) The first kernel processing at an $I \cdot N$ dimension. (c) The second kernel working at an $O \cdot N$ dimension.

Algorithm 7 Unrolled base conversion

function `Conv_Scalar`($\boldsymbol{c}', \boldsymbol{c}, params$)
  $\mathsf{tid} \in [0, \lceil I \cdot N/F \rceil)$, $i := \mathsf{tid}/(N/F)$, $f \in [0, F)$
  $(\tilde{q}_i, q_i, \mu_i) \leftarrow params[i]$, $\{c_f\} := \texttt{WideLoad}(\boldsymbol{c}^{(i)})$
  return $\boldsymbol{c}' \leftarrow \texttt{WideStore}(\texttt{CModMul}(\{c_f\}, \tilde{q}_i, q_i, \mu_i))$
function `Conv_Matrix`($\boldsymbol{c}'', \boldsymbol{c}', params$)
  $\mathsf{tid} \in [0, \lceil O \cdot N/F \rceil)$, $f \in [0, F)$, $\{acc_f\} := \{0\}$
  $\mathbb{S} \leftarrow params$ ▷ Cache in SMEM
  for $i \in [0, I)$ **do**
    $\{c_f\} := \texttt{WideLoad}(\boldsymbol{c}'^{(i)})$
    $q_i^* \leftarrow \mathbb{S}[I \cdot [\mathsf{tid}]_O + i]$
    $\{acc_f\} := \{acc_f\} + c_f \cdot q_i^*$
  $(r_k, \mu_k) \leftarrow \mathbb{S}[[\mathsf{tid}]_O]$
  return $\boldsymbol{c}'' \leftarrow \texttt{WideStore}(\texttt{ModRed}(\{acc_f\}, r_k, \mu_k))$

the norm is bound by I, we pre-compute the multiples of Q in a look-up table for selecting results based on $\boldsymbol{v}$.

Optimization with Loop Unroll. Performance of the two kernels can be hindered by memory constraints due to excessive loading and storing, particularly when constants are repeatedly loaded by each thread. To mitigate this, we apply loop unrolling, as demonstrated in Algorithm 7. We set an unroll factor F, wherein each thread processes F coefficients of I residues. This reduces the loading of pre-computed values (q_i^*, r_k, μ_k) from F to one and allows 64-bit operations to be replaced with wider instructions, hence reducing pipeline overhead. In Algorithm 7, we use `WideLoad` and `WideStore` to denote general case wide load and store instructions, like the `ldg` intrinsic. For example, with $F = 2$, two 64-bit load instructions can be replaced by a single 128-bit one.

D. Decryption

To cater to both the BEHZ and HPS techniques, we design a generic module, depicted in Fig. 3. The decryption in the BFV scheme first computes the inner product of $\mathbf{sk}$ and $\mathbf{ct}$, followed by scaling the result by $\frac{t}{Q}$ to the nearest integer, where the later is the main difference between BEHZ and HPS.

In the first step, the inner product of $\mathbf{sk}$ and $\mathbf{ct}$ is computed. Since relinearization is optional, $\mathbf{ct}$ may contain more than two elements, $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \ldots)$. In such cases, a generic inner product with $\mathbf{sk} := (1, \boldsymbol{s}, \boldsymbol{s}^2, \ldots)$ is required. Thus, we design an accumulated multiplication kernel. We first invoke

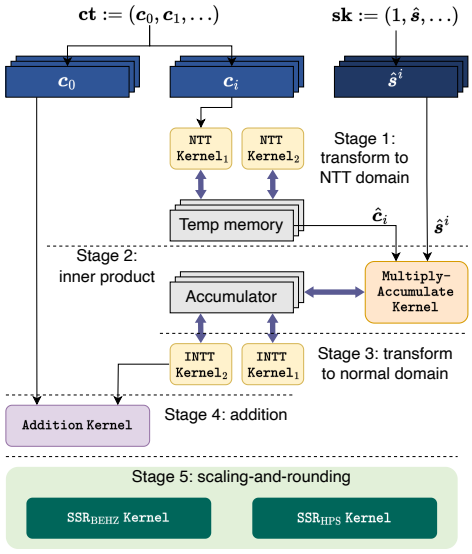

Fig. 3. The architecture of generic decryption module designed to accommodate both BEHZ and HPS techniques.

Algorithm 8 $\mathrm{SSR}_{\mathrm{BEHZ}}$: BEHZ simple scaling-and-rounding

Input: $\boldsymbol{x} := \langle \mathbf{ct}, \mathbf{sk} \rangle$
Output: $\boldsymbol{m}$
$\mathsf{tid} \in [0, L \cdot N)$, $i := \mathsf{tid}/N$
$\boldsymbol{x}'[\mathsf{tid}] := \texttt{ModMul}(\boldsymbol{x}[\mathsf{tid}], [t\gamma]_{q_i}, q_i)$
$\boldsymbol{f}' := \texttt{Conv\_Scalar}(\boldsymbol{x}', Q)$
$\boldsymbol{f} := \texttt{Conv\_Matrix}(\boldsymbol{f}', Q, \{t, \gamma\})$
$\mathsf{tid} \in [0, N)$
$\boldsymbol{f}^{(\gamma)}[\mathsf{tid}] := \texttt{ModRed}(\gamma - \boldsymbol{f}^{(\gamma)}[\mathsf{tid}], \gamma)$
$\boldsymbol{f}^{(t)}[\mathsf{tid}] := \texttt{ModRed}(\boldsymbol{f}^{(t)}[\mathsf{tid}] + \boldsymbol{f}^{(\gamma)}[\mathsf{tid}], t)$
$\boldsymbol{m}[\mathsf{tid}] := \texttt{ModMul}(\boldsymbol{f}^{(t)}[\mathsf{tid}], [\gamma^{-1}]_t)$

the NTT kernels to transition elements in $\mathbf{ct}$ except $\boldsymbol{c}_0$ to the NTT domain for efficient multiplication with $\boldsymbol{s}^i$. We establish a buffer in GMEM for memory reuse. The accumulated result is then converted back and added to $\boldsymbol{c}_0$. The subsequent scaling-and-rounding phase depends on the specific method used. Our implementations of the two methods, denoted as $\mathrm{SSR}_{\mathrm{BEHZ}}$ and $\mathrm{SSR}_{\mathrm{HPS}}$, respectively, are outlined in Algorithms 8 and 9.

Optimizations to BEHZ Method. Unlike the simpler HPS, the BEHZ involves four complex stages with two levels of parallelism. The first two kernels conduct modular multiplication with γt and the initial kernel of conversion `Conv_Scalar` are scalar modular multiplication. Therefore, we fuse them into a single kernel with $L \cdot N/F$ threads, eliminating GMEM interactions $\boldsymbol{x}'$. Next, we launch the second kernel of base conversion, `Conv_Matrix`, which utilizes $\lceil K \cdot N/F \rceil$ threads for unrolled modular multiplication with the matrix. The following process requires only coefficient-wise computation, thus we optimize it to one kernel for better memory pattern.

E. Homomorphic Multiplication

Homomorphic multiplication in the BFV scheme is a major computational challenge. We aim to accelerate all four variants, each with distinct merits and demerits, to develop comprehensive and efficient solutions.

Algorithm 9 $\texttt{SSR}_{\text{HPS}}$:HPS simple scaling-and-rounding

Input: $\boldsymbol{x} := \langle \mathbf{ct}, \mathbf{sk} \rangle$
Output: $\boldsymbol{m}$
$\mathsf{tid} \in [0, N)$
$sum_I := 0, sum_F := 0.0$
for $i \in [0, L)$, $j \in [0, d_s)$ **do**
  $\{x^{(i)}_{\mathsf{tid},j}\} \leftarrow \boldsymbol{x}[iN + \mathsf{tid}]$
  $sum_F = sum_F + \sum_{j=0}^{d_s-1} x^{(i)}_{\mathsf{tid},j} \tilde{\theta}_{i,j}$
  $sum_I = sum_I + \sum_{j=0}^{d_s-1} \texttt{ModMul}(x^{(i)}_{\mathsf{tid},j} \omega_{i,j})$
$sum_F = sum_F + sum_I$
$sum = \lfloor sum_F - t \cdot \lfloor sum_F \cdot \frac{1}{t} \rfloor \rceil$
$\boldsymbol{m}[\mathsf{tid}] := sum$

Algorithm 10 $\texttt{CSR}_{\text{HPS}}$: HPS complex scaling-and-rounding

Input: $\boldsymbol{c} := \{\boldsymbol{c}^{(i)}\}$, $i \in [0, I)$
Output: $\boldsymbol{c}' := \{\boldsymbol{c}'^{(i)}\}$, $i \in [0, O)$
$\mathsf{tid} \in [0, N)$
$sum_I := \{0, 0\}, sum_F := 0.0$
for $k \in [O, I)$ **do**
  $sum_F := sum_F + \boldsymbol{c}[kN + \mathsf{tid}] \cdot \tilde{\theta}_{k-O}$
for $k \in [0, O)$ **do**
  for $j \in [O, I)$ **do**
    $sum_I := sum_I + \boldsymbol{c}[jN + \mathsf{tid}] \cdot \omega_{k,j-O}$
  $sum_I := sum_I + \boldsymbol{c}[kN + \mathsf{tid}] \cdot \lambda_k$
  $\boldsymbol{c}'[kN + \mathsf{tid}] = \texttt{ModRed}(\texttt{ModRed}(sum_I) + \lfloor sum_F \rceil)$
return $\boldsymbol{c}'$

Existing researches mainly concentrate on enhancing the BEHZ variant. For instance, the study [20] focuses on small parameters and employs 10 kernels, leaving room for potential computational and memory optimizations. The research [21] proposes compact designs by dividing the process into three stages for kernel fusing, but there remain unexplored optimizations for of BEHZ-type multiplication. The HPS variant has received relatively less attention. The work [22] limits the data type to 32-bit arithmetic, restricting its applicability.

In contrast, our design strategy addresses these limitations by accommodating more generalized cases and proposing novel enhancements for both schemes. Below, we detail our implementations and optimizations.

1) Optimizations in BEHZ-type Tensoring: We have implemented several key optimizations to enhance the efficiency of BEHZ-type tensoring. These include:

- **Intra-Arithmetic Fusion:** Observing that `Conv_Scalar` comprises scalar multiplication and `Conv_Matrix` is frequently followed by arithmetic operations in the same bases, we investigate several fusion strategies for performance enhancement. First, we adapt multiplying constant into `Conv_Scalar`. Then, we incorporate $\tilde{m}$ into the precomputation of the conversion matrix to directly derive $[\tilde{m}\boldsymbol{c}]_Q$, thus eliminating modular multiplication with $\tilde{m}$. Furthermore, we fuse multiply-and-subtract computation into `Conv_Matrix` in both the fast flooring and `ConvSK` stages, as these computations use the same RNS bases. These optimizations have significantly minimized time-consuming GMEM access of the polynomials.
- **Inter-Conversion Fusion:** In BEHZ, it is common to convert one polynomial to two different bases successively, which involves calling two base conversions. Here, the `Conv_Scalar` phase is the same and can be reused. Observing that, we fuse these two operations, thus saving one stage of the conversions. This technique is applied in two cases, first is for $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}\to\mathcal{B}_{sk}}$ and $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}\to\{\tilde{m}\}}$, and the second is for `ConvSK`, where $[\tilde{\boldsymbol{c}}_h]_{\mathcal{B}}$ are converted to base $\{m_{sk}\}$ and base $\mathcal{B}$.

2) Generic HPS-type Tensoring Design: We develop a generic HPS-type tensoring framework compatible with all three HPS variants. For two input ciphertexts $\mathbf{ct}_1, \mathbf{ct}_2 \in \mathcal{R}_Q$, the framework extends the base of $\mathbf{ct}_1$ to $\mathcal{Q} \cup \mathcal{B}$ or scales it down to $\mathcal{Q}_l$ before extending to $\mathcal{Q}_l \cup \mathcal{B}_l$. According to the three variants, $\mathbf{ct}_2$ is extended to $\mathcal{Q} \cup \mathcal{B}$, converted to $\mathcal{B}$ then to $\mathcal{Q} \cup \mathcal{B}$, or convert to $\mathcal{B}_l$ and then to $\mathcal{Q}_l \cup \mathcal{B}_l$. After the tensor product, the ciphertexts are converted back to the original base $\mathcal{Q}$. Each variant varies in base size, thereby consuming different computation and memory resources. Considering the frequent invocation of the HPS-type complex scaling-and-rounding function with varying input and output sizes, we have designed a generic `CSR` kernel to accommodate all cases, as outlined in Algorithm 10. We implement lazy reduction, performing modular reduction only after accumulation. An 128-bit accumulator, sum_I, handles cases without overflow. This versatile approach enhances efficiency and flexibility.

3) Reducing Computation in LHPS: In LHPS, an approach analogous to the tensoring stage is employed wherein the ciphertexts are scaled down to modulo Q_l for relinearization. Remarkably, the final step in $\texttt{Tensor}_{LHPS}$ involves scaling up the ciphertexts from modulo Q_l to Q. Consequently, we adopt an optimization by fusing the tensoring and relinearization stages in this variant. This streamlined approach yields multiple benefits. Firstly, if the level remains consistent in the two stages, the dual scaling operations effectively cancel each other out, thereby reducing computational load and eliminating noise typically introduced by these operations. This innovative strategy offers a significant enhancement in computational efficiency and precision, reinforcing the efficacy of this variant.

F. Complexity Analysis

Below, we analyze the BFV variants built on our framework. We start by detailing the necessary pre-computations for these variants and then discuss their computational complexity. Some results are summarized in Tables I and II.

1) Pre-computation: Pre-computation is essential for expediting subsequent calculations. For every modulus q, we perform several pre-computations, denoted as $\mathsf{q_p rm}$. Specifically, we compute $\mu := \lfloor \frac{2^{64}}{q} \rfloor$ for fast modular reduction, and two sequences of ζ^i and $(\zeta^{-1})^i$ as the NTT and INTT tables, where $\zeta^{2N} \equiv 1 \bmod q$. Additionally, we calculate $\mu_{0,i}$ and $\mu_{1,i}$ for the rapid const modular reduction `CModMul`. These calculations cumulatively require $4N + 1$ integers.

For both BEHZ and HPS base conversion, we pre-compute the convert matrices $[\tilde{q}_i]_{q_i}$ and $[q_i^*]_{r_k}$ for conversions from base $\mathcal{Q} := \{q_i\}$ to $\mathcal{B} := \{r_k\}$, where $i \in [1, L]$ and $k \in [1, K]$. For HPS, additional computations of $[v \cdot Q]_{r_k}$ are needed where $v \in [0, L]$. The total cost amounts to $L(K + 1)$ for the BEHZ method and $2L + LK + 1$ for the HPS method.

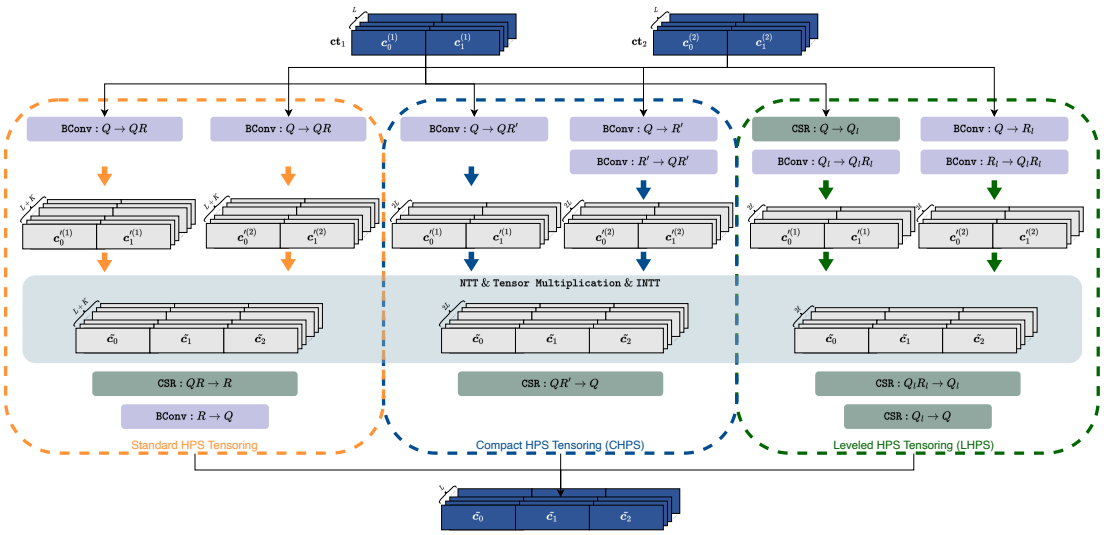

Fig. 4. A unified module for original HPS, compact HPS, and leveled HPS variants.

TABLE I
PRE-COMPUTATIONS FOR ALL VARIANTS. NOTATION $Q_l := \Pi_{i=1}^{l} q_i$ AND $R_l := \Pi_{k=1}^{l} r_k$ ARE USED TO ENSURE COMPATIBILITY WITH CHPS AND LHPS, AND $s_\iota \in \{q_i, r_k\}$. FOR CHPS, $l := L$. FOR LHPS, THESE PRE-COMPUTATIONS ARE PERFORMED FOR EACH LEVEL $l \leq L$.

Operation	Values	Description	Size	Type
BEHZ Decryption	$[\gamma^{-1}]_t$ $[t \cdot \gamma]_{q_i}$ $[-Q^{-1}]_{\{t,\gamma\}}$ $\texttt{Conv}^{\texttt{BEHZ}}_{\mathcal{Q}\to\{t,\gamma\}}$	$\gamma^{-1} \bmod t$ $t \cdot \gamma \bmod q_i$ $-Q^{-1} \bmod \{t,\gamma\}$ Base Convertor	1 L 2 $3L$	Integer
HPS Decryption	$\omega_{i,j}$ $\theta_{i,j}$	$[t \cdot [[(Q/q_i)^{-1}]_{q_i} \cdot B^j]_{q_i}/q_i]_t$ $[t \cdot [[(Q/q_i)^{-1}]_{q_i} \cdot B^j]_{q_i}]_{q_i}/q_i$	$d_s \cdot L$ $d_s \cdot L$	Integer Double
BEHZ Tensoring	q_prm $[t]_{\mathcal{B}_{sk}}$, $[Q]_{\mathcal{B}_{sk}}$, $[Q^{-1}]_{\mathcal{B}_{sk}}$, $[\tilde{m}^{-1}]_{\mathcal{B}_{sk}}$ $[R]_{q_i}$ $[R^{-1}]_{m_{sk}}$, $[-Q^{-1}]_{\tilde{m}}$ $\texttt{Conv}^{\texttt{BEHZ}}$	Auxiliary $\mathcal{B}_{sk}$ $\{t, Q, Q^{-1}, \tilde{m}^{-1}\} \bmod \{r_k, m_{sk}\}$ $R \bmod q_i$ $R^{-1} \bmod m_{sk}$, $-Q^{-1} \bmod \tilde{m}$ $\mathcal{Q} \to \mathcal{B}_{sk}$, $\mathcal{Q} \to \tilde{m}$, $\mathcal{B} \to \mathcal{Q}$, $\mathcal{B} \to m_{sk}$	$(4N+1)(K+1)$ $4(K+1)$ L 2 $2LK+5L+3K+3$	Integer
All HPS Tensoring	q_prm $\texttt{Conv}^{\texttt{HPS}}$	Auxiliary $\mathcal{B}$ $\mathcal{Q} \to \mathcal{B}$, $\mathcal{B} \to \mathcal{Q}$	$(4N+1)K$ $2(KL+K+L+1)$	Integer
HPS	ω_i', λ_k θ_i	$[t \cdot R \cdot [(QR/s_\iota)^{-1}]_{s_\iota}/s_\iota]_{r_k}$ $[t \cdot R \cdot [(QR/q_i)^{-1}]_{q_i}]_{q_i}/q_i$	$K(L+1)$ L	Integer Double
CHPS & LHPS	ω_i', λ_k θ_i	$[t \cdot Q_l \cdot [(Q_l R_l/s_\iota)^{-1}]_{s_\iota}/s_\iota]_{q_i}$ $[t \cdot Q_l \cdot [(Q_l R_l/r_i)^{-1}]_{r_i}]_{r_i}/r_i$	$L(K+1)$ K	Integer Double
LHPS	ω_i'', λ_k' θ_i''	$[Q_l \cdot [(Q/q_\tau)^{-1}]_{q_\tau}/q_\tau]_{q_\tau}$ $Q_l \cdot [(Q/q_\tau)^{-1}]_{q_\tau}/q_\tau$	$l \cdot (L-l+1)$ $L-l$	Integer Double

2) Computational Complexity: We present an overview of the computational complexity of our approach in Table II. Prior studies [14], [22] perform analysis in operation level by counting numbers include modular operation, floating-point operation, NTT, and base conversion. Our analysis provides a more detailed evaluation tailored to the GPU platform. Since GPU-based HE implementations are often memory-bound, we consider the memory access complexity of each variant, including GMEM load, GMEM store, and SMEM access. We also detail the count of integer multiplication (IM) and floating-point (FP) instructions, noting that modular reduction instruction counts vary with data sizes.

Besides, we compare our implementation complexity with related works. For BEHZ, we reference [20]. In `NTT`, their approach requires GMEM iterations at each level, resulting in $\mathcal{O}(NL\log N)$ GMEM access for a polynomial with L moduli. Our implementation reduces this to $\mathcal{O}(\lceil \frac{N}{3} \log N \rceil \cdot L)$ by using SMEM. For base conversion, we reduce GMEM access for precomputed values by a factor of F through unrolling, and ciphertext access by a factor of 2 through wider load and store instructions. Our fusing technique further reduces GMEM access and the total number of base conversions. For HPS, [22] supports only 32-bit arithmetic with limited precision. Our implementation provides more variants, functionality, and better precision, albeit with more instructions.

V. PERFORMANCE EVALUATION

In this section, we present the performance of our implementation and compare it with related works.

A. Experimental Setup

We compile the C/C++ code using g++ 12.2.0 and the GPU implementations with CUDA 11.8 on an Arch Linux system

TABLE II
COMPARISON OF COMPUTATIONAL COMPLEXITY ACROSS VARIANTS USING OUR FRAMEWORK. THE METRICS ARE EVALUATED BASED ON THE NUMBER OF INSTRUCTIONS.

Operations	Memory Instructions			Arithmetic Instructions	
	GMEM load	GMEM store	SMEM load	IM	FP
$\texttt{SSR}_{\text{BEHZ}}$	$\mathcal{O}((3L/F+L+3)N+2L+4)$	$\mathcal{O}((2L+4)N)$	$\mathcal{O}(2LN/F)$	$\mathcal{O}(8LN+22N)$	0
$\texttt{SSR}_{\text{HPS}}$	$\mathcal{O}(LN+2d_sL+5L/F+2L+2)$	$\mathcal{O}(N)$	0	$\mathcal{O}(d_sLN)$	$\mathcal{O}((d_sL+1)N)$
$\texttt{Tensor}_{\text{BEHZ}}$	$\mathcal{O}((35L+45K+45)N/F$ $+10KLN+95LN+96KN$ $+4KL+8L+7K+9)$	$\mathcal{O}(20(L+K+1)N+5LN$ $+8(K+1)N+3N)$	$\mathcal{O}(7\lfloor\log N/3\rfloor(L+K+1)N)$ $+(10LK+10L+6K+6)N/F$	$\mathcal{O}(7(L+K+1)N\log N)+212N$ $+164KN+109LN+20LKN$	0
$\texttt{Tensor}_{\text{HPS}}$	$\mathcal{O}((5K+5L)N/F+16KLN$ $+94LN+80KN+7KL$ $+37L+36K)$	$\mathcal{O}(29LN+20.5KN)$	$\mathcal{O}(7\lfloor\log N/3\rfloor\cdot(L+K)N$ $+7LKN/F)$	$\mathcal{O}(7(L+K)N\log N$ $+18KLN+56LN+92KN)$	$\mathcal{O}(7LKN+3LN+3KN)$
$\texttt{Tensor}_{\text{CHPS}}$	$\mathcal{O}(30LN/F+15L^2N+166LN$ $+4KN+6L^2+74L)$	$\mathcal{O}(47LN)$	$\mathcal{O}(14LN\lfloor\log N/3\rfloor+6L^2N/F)$	$\mathcal{O}(14LN\log N$ $+162LN+18L^2N)$	$\mathcal{O}(4L^2N+6LN)$
$\texttt{Tensor}_{\text{LHPS}}$	$\mathcal{O}(5lN/F+7l^2N+10LN$ $+4KN+161lN+8lLN$ $+2lL+82l+4l^2)$	$\mathcal{O}(40lN+LN)$	$\mathcal{O}(14lN\lfloor\log N/3\rfloor+6l^2N/F)$	$\mathcal{O}(14lN\log N+10l^2N$ $+165lN+4LN+8lLN)$	$\mathcal{O}(4l^2N+2LN+4lN)$

TABLE III
PERFORMANCE OF OUR OPTIMIZED BFV VARIANTS IMPLEMENTATION ON BOTH THE 3090 TI GPU AND A100 GPU (MEASURE IN MILLISECONDS). THE SPEEDUP FACTOR IS DETERMINED BY COMPARING PERFORMANCE OF OUR IMPLEMENTATION ON A100 GPU AGAINST THE PERFORMANCE OF OPENFHE WITH AN EQUIVALENT PARAMETER SET RUNNING ON A MULTI-THREADED CPU.

Operations	Our work										OpenFHE [7]	Speedup
	3090 Ti					A100					Multi-thread CPU	
$\log N$	13	14	14	15	15	13	14	14	15	15	14	
$\log Q$	120	300	240	660	540	120	300	240	660	540	300	
$\log P$	60	60	120	60	180	60	60	120	60	180	60	
$\texttt{HMult}_{\text{BEHZ}}$	0.311	0.463	0.398	1.378	0.921	0.486	0.634	0.573	1.426	1.017	20.200	**31.9**×
$\texttt{HMult}_{\text{HPS}}$	0.207	0.389	0.289	1.445	0.968	0.310	0.466	0.395	1.293	0.872	14.400	**30.9**×
$\texttt{HMult}_{\text{CHPS}}$	0.189	0.323	0.261	1.302	0.836	0.292	0.445	0.367	1.207	0.794	11.800	**26.5**×
$\texttt{HMult}_{\text{LHPS}}$	-	0.290	0.255	1.187	0.775	-	0.408	0.318	1.145	0.735	11.400	**27.9**×
$\texttt{Dec}_{\text{BEHZ}}$	0.039	0.049	0.046	0.104	0.086	0.066	0.072	0.071	0.108	0.100	1.030	**14.3**×
$\texttt{Dec}_{\text{HPS}}$	0.032	0.040	0.037	0.092	0.075	0.052	0.058	0.057	0.090	0.085	0.960	**16.6**×

TABLE IV
PERFORMANCE COMPARISON OF OUR OPTIMIZED BEHZ VARIANT GPU IMPLEMENTATION WITH RELATED WORKS [20]–[22] (MEASURE IN MILLISECONDS). SPEEDUP RATIOS ARE DETERMINED BY COMPARING PERFORMANCE WITH EACH RESPECTIVE WORK ON THE SAME HARDWARE PLATFORM.

Operations	Our work							[20]		[21]	[22]
	3090 Ti			A100			3060 Ti	3060 Ti	Speedup	V100S	V100
$\log N$	13	14	15	13	14	15	15	15		15	15
$\log QP$	218	438	881	218	438	881	881	881		881	600
$\texttt{Tensor}_{\text{BEHZ}}$	0.261	0.380	1.037	0.400	0.489	1.068	2.267	3.757	**39.7%**	1.608	5.705
Relin	0.100	0.212	0.988	0.147	0.293	0.983	2.209	3.150	**29.9%**	-	
$\texttt{Dec}_{\text{BEHZ}}$	0.040	0.054	0.131	0.067	0.075	0.129	0.317	-		-	0.134

with kernel 5.15. For establishing a CPU baseline, we use an Intel(R) Core(TM) i9-12900KS CPU, equipped with 16 cores. For our GPU implementation, we test the performance across two different GPUs. These include a NVIDIA Tesla A100 80G PCIe and a NVIDIA GeForce RTX 3090 Ti, providing a diverse range of computational capabilities for our analysis. We evaluate the procedures performance in terms of execution time, reported as the mean of 100 executions in milliseconds (ms). Note that the reported times also account for the latency associated with data transfer between the CPU and GPU.

The parameter sets used in our performance evaluation all achieve 128-bit security [31]. When comparing with the CPU baseline, we apply a widely used configuration by setting each modulus to 60 bits. For comparisons with related work, we use the same parameter sets as those studies for a fair comparison.

B. Performance of BFV Variants

Table III elucidates the performance outcomes of our tailored BFV variant implementations across two distinct GPUs, measured in milliseconds. The homomorphic multiplication HMult comprises both tensoring and relinearization, while Dec corresponds to decryption. For the leveled variant $\texttt{HMult}_{\text{LHPS}}$, we present the performance resulting from a single-level drop, except for $N = 2^{13}$ parameter sets where no level drop is allowed. We also compare our results with the OpenFHE [7] running on a multi-threaded CPU to establish a baseline.

Our implementations demonstrate significant speedups compared to the OpenFHE, with the speedup factors range from 14.3× to 31.9×. On the 3090 Ti GPU, the CHPS and LHPS variants range from 0.189 ms to 1.302 ms and 0.290 ms to 1.187 ms, respectively, demonstrating the utility of these specialized variants in enhancing computational efficiency. The LHPS variant shows promising performance, making it as a highly efficient solution for certain use cases, with clear benefits over the traditional BFV implementation. Within our experimental landscape, the HPS variants outperform the BEHZ variant in most cases. For instance, on the 3090 Ti GPU, with parameters set to $\log N = 14$, $\log Q = 300$, and $\log P = 60$, the HMult operation showcases an approximately 19% improvement of HPS over BEHZ.

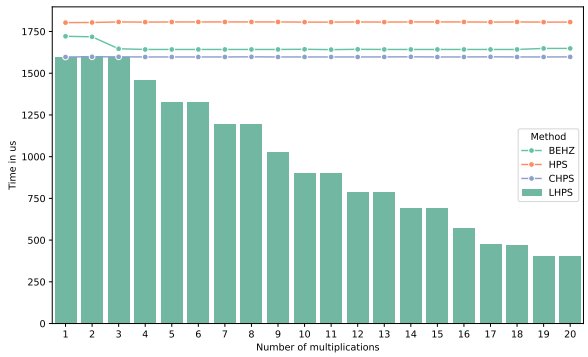

Fig. 5. Performance comparison of four homomorphic multiplication variants within a recursive binary tree application scenario, assessed across varying multiplication depths.

C. Comparison with Related Works

Given that our GPU-based implementation of the HPS variants is the first of its kind, we have chosen to benchmark the performance of our BEHZ variant against several relevant studies [20]–[22]. The work in [20] represents the most recent and advanced implementation of the BEHZ variant on a GPU. Meanwhile, [22] illustrates a GPU implementation utilizing 32-bit arithmetic.

The performance of our optimized BEHZ variant implementation on different GPU platforms and a comprehensive comparison is provided in Table IV. As the work of [20] remains closed-source, we compared the performance to ours on an equivalent hardware platform. When evaluating the $\log N = 15$ and $\log QP = 881$ parameter set, our $\texttt{Tensor}_{\text{BEHZ}}$ and `Relin` operations show execution times of 2.267ms and 2.209ms, respectively. This corresponds to substantial speedup ratios of 39.7% and 29.9% respectively, thereby underscoring the efficacy of our GPU-accelerated implementation.

D. Sensitivity in Application

To assess the impact of different homomorphic multiplication variants in a real-world application, we implemented an application performing recursive homomorphic multiplication. In our experiment, we select two ciphertexts, $\mathbf{ct}_1$ and $\mathbf{ct}_2$, and recursively multiplied $\mathbf{ct}_1$ by $\mathbf{ct}_2$ 20 times, updating $\mathbf{ct}_1$ with each result. We set the parameters to $\log N = 15$, $\log QP = 840$, with one special modulus.

Throughout the evaluation, the performance remained constant for the BEHZ, HPS, and CHPS variants. However, the LHPS variant showed a different pattern, with layers gradually dropping, leading to improved performance over time. After 20 multiplications, the LHPS variant demonstrated approximately $4\times$ better performance compared to the other variants. The data underscores the impressive performance of the LHPS variant in specific application scenarios. This suggests that the LHPS variant could be an exceptionally efficient alternative for certain use cases, thereby offering discernible advantages over traditional BFV implementations.

VI. Conclusion

In this work, we provide a comprehensive and insightful study on accelerating and comparing BFV variants on GPUs. Our research not only develops a universal framework that accommodates all BFV variants, but also presents several notable advancements, including support for large-parameter HPS variants on GPU and significant optimizations that minimize computational and memory consumption. Performance evaluation shows substantial speed improvements, with our implementation of the leveled HPS variant emerging as a promising solution for specific applications. Our work contributes to the ongoing advancements in the field of Homomorphic Encryption and provides avenues for future explorations and improvements in privacy-preserving computations.

Acknowledgments

This work is supported by the National Key R&D Program of China (Grant No. 2022YFB2702000), the National Natural Science Foundation of China (Grant No. 62132008, 62372417, 62071222), and the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20220075).

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, 2009.
[2] ——, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.
[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in Innovations in Theoretical Computer Science 2012, 2012, pp. 309–325.
[4] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in Advances in Cryptology - CRYPTO 2012, ser. Lecture Notes in Computer Science, vol. 7417, 2012, pp. 868–886.
[5] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," IACR Cryptol. ePrint Arch., p. 144, 2012.
[6] "Microsoft SEAL (release 4.0)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.
[7] "OpenFHE," https://github.com/openfheorg/openfhe-development, Jan. 2023.
[8] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," 2019, pp. 812–821.
[9] Q. Lou, W. Lu, C. Hong, and L. Jiang, "Falcon: Fast spectral inference on encrypted data," 2020.
[10] W.-j. Lu, J.-j. Zhou, and J. 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, 2018, pp. 67–74.
[11] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from fully homomorphic encryption with malicious security," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1223–1237.
[12] J. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of fv like somewhat homomorphic encryption schemes," in Selected Areas in Cryptography - SAC 2016, ser. Lecture Notes in Computer Science, 2017, pp. 423–442.
[13] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in Topics in Cryptology - CT-RSA 2019, ser. Lecture Notes in Computer Science, vol. 11405, 2019, pp. 83–105.
[14] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," in Advances in Cryptology - ASIACRYPT 2021, ser. Lecture Notes in Computer Science, 2021, pp. 608–639.
[15] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A cuda-accelerated word-wise homomorphic encryption library," IEEE Transactions on Dependable and Secure Computing, pp. 1–12, 2024.

[16] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2018, no. 2, pp. 70–95, 2018.
[17] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," The Journal of Supercomputing, 2021.
[18] P. G. M. R. Alves, J. N. Ortiz, and D. F. Aranha, "Faster homomorphic encryption over GPGPUs via hierarchical DGT," in Financial Cryptography and Data Security, ser. Lecture Notes in Computer Science, 2021, pp. 520–540.
[19] E. R. Türkoglu, A. S. Özcan, C. Ayduman, A. C. Mert, E. Öztürk, and E. Savas, "An accelerated GPU library for homomorphic encryption operations of BFV scheme," in 2022 IEEE International Symposium on Circuits and Systems (ISCAS), 2022, pp. 1155–1159.
[20] A. S. Özcan, C. Ayduman, E. R. Türkoglu, and E. Savas, "Homomorphic encryption on GPU," IACR Cryptol. ePrint Arch., vol. 2022, p. 1222, 2022.
[21] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "CARM: CUDA-accelerated RNS multiplication in word-wise homomorphic encryption schemes for internet of things," IEEE Trans. Computers, vol. 72, no. 7, pp. 1999–2010, 2023. [Online]. Available: https://doi.org/10.1109/TC.2022.3227874
[22] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," IEEE Transactions on Emerging Topics in Computing, vol. 9, no. 2, pp. 941–956, 2021.
[23] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. Khin Mi Mi, "Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 2, pp. 379–391, 2021.
[24] A. Al Badawi, C. Jin, J. Lin, C. F. Mun, S. J. Jie, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "Towards the alexnet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs," IEEE Transactions on Emerging Topics in Computing, vol. 9, no. 3, pp. 1330–1343, 2021.
[25] M. R. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. E. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption standard," IACR Cryptol. ePrint Arch., vol. 2019, p. 939, 2019.
[26] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in Advances in Cryptology - CRYPTO 2012, ser. Lecture Notes in Computer Science, vol. 7417, 2012, pp. 850–867.
[27] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in Topics in Cryptology - CT-RSA 2020, ser. Lecture Notes in Computer Science, 2020, pp. 364–390.
[28] V. Shoup, "NTL: A library for doing number theory," https://libntl.org, 2001.
[29] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in IEEE International Symposium on Workload Characterization, IISWC 2020, 2020, pp. 264–275.
[30] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2021, no. 4, pp. 114–148, 2021.
[31] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," Journal of Mathematical Cryptology, vol. 9, no. 3, pp. 169–203, 2015.

Shiyu Shen is a PhD candidate in School of Computer Science, Fudan university. Her research interests include lattice-based cryptography, homomorphic encryption, and cryptographic engineering. Her email address is shenshiyu21@m.fudan.edu.cn.

Hao Yang is a PhD candidate at College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics. His research interests include homomorphic encryption, lattice-based cryptography, and cryptographic engineering. His email address is crypto@d4rk.dev.

Wangchen Dai received the B.Eng. degree in electrical engineering and automation from Beijing Institute of Technology, China, in 2010, the M.A.Sc. degree in electrical and computer engineering from the University of Windsor, Canada, in 2013, and the Ph.D. degree in electronic engineering from the City University of Hong Kong in 2018. After completing the Ph.D. study, he had appointments at Hardware Security Lab, Huawei Technologies Company Ltd., in 2018, and the Department of CSSE, Shenzhen University in 2020, respectively. His research interests include cryptographic hardware and embedded systems, fully homomorphic encryption, and reconfigurable computing.

Lu Zhou is a professor in Nanjing University of Aeronautics and Astronautics, China. She received the B.S. and M.S. degrees in computer science from Shandong University in 2012 and 2015, and the Ph.D degree in computer science from University of Aizu in 2019, respectively. Her main research interests include cryptographic and security solutions for the Internet of Things. Her research is supported by the National Natural Science Foundation of China and the Natural Science Foundation of Jiangsu Province.

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

Yunlei Zhao received his PhD at Fudan University in 2004. He is now a distinguished professor at Fudan university. His main research interests include post-quantum cryptography, cryptographic protocols, theory of computing. His email address is ylzhao@fudan.edu.cn.