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 facilitating computations on encrypted data, opening new paths for privacy-focused computations. The Brakerski-Fan-Vercauteren (BFV) scheme, a promising HE scheme, raises considerable performance challenges. Graphics Processing Units (GPUs), with considerable parallel processing abilities, have emerged as an effective solution.**

In this work, we present an in-depth study focusing on accelerating and comparing BFV variants on GPUs, including Bajard-Eynard-Hasan-Zucca (BEHZ), Halevi-Polyakov-Shoup (HPS), and other recent variants. We introduce a universal framework accommodating all variants, propose optimized BEHZ implementation, and first support HPS variants with large parameter sets on GPUs. Moreover, we devise several optimizations for both low-level arithmetic and high-level operations, including minimizing instructions for modular operations, enhancing hardware utilization for base conversion, implementing efficient reuse strategies, and introducing intra-arithmetic and inner-conversion fusion methods, thus decreasing the overall computational and memory consumption.

Leveraging our framework, we offer comprehensive comparative analyses. Our performance evaluation showcases a marked speed improvement, achieving 31.9× over OpenFHE running on a multi-threaded CPU and 39.7% and 29.9% improvement, respectively, over the state-of-the-art GPU BEHZ implementation. Our implementation of 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 of integer vectors. 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 privacy inference [8], [9], decision tree evaluation [10], and set intersection [11].

In spite of the substantial benefits of the BFV scheme, its performance is still far from satisfactory. Unlike the leveled structure of BGV, BFV adheres to a scale-invariant design with a constant ciphertext modulus. This results in homomorphic operations being conducted on ciphertexts with a large modulus size, leading to inefficiencies due to the enormous data magnitude. Additionally, the encryption structure of BFV adds complexity to the decryption and multiplication processes due to the division-and-rounding operation. Since approximate arithmetic is incompatible with the Residue Number System (RNS) representation, this operation has been considered difficult to perform on the RNS variant of BFV.

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

However, the high computational demands of these methods pose a significant barrier to developing practical HE-based applications. GPU acceleration, explored by numerous works, is crucial to accelerating BFV and its applications. Badawi et al. [15] 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 [16]–[20], 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 [21], but was limited to 32-bit arithmetic due to precision constraints. More scalable implementations later emerged, extending to multi-GPU architectures [22]. Efforts to accelerate privacy-preserving applications based on BFV have

S. Shen and Y. Zhao are with the School of Computer Science, Fudan University, Shanghai, China.

H. Yang and L. Zhou are with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics.

W. Dai and Z. Liu are with the Zhejiang Lab.

Manuscript received July, 2023.

also been researched, such as homomorphic convolutional neural networks (HCNN) [23] and gradient boosting inference (XGBoost) [18], [19]. While significant progress have been made, several challenges still persist. There is potential for further performance enhancement in current implementations, and present GPU implementation of HPS is limited by the parameter configurations it supports. Furthermore, the acceleration of recent variants with potentially superior performance remains 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. Specifically, we offer optimized GPU implementations of the BEHZ variant and first support the HPS with large parameter sets and other HPS variants. Our optimizations include:

- For arithmetic operations, we implement modular operations with minimal instructions for different data sizes. We propose optimized base conversion implementation with high hardware utilization, and reduce memory access through loop unrolling.
- For decryption, we designed a finely-tuned memory reuse strategy and optimize memory access patterns to decrease consumption and latency.
- For BEHZ multiplication, we propose intra-arithmetic fusion and inner-conversion fusion methods, allowing to reuse temporary values, decrease the number of base conversions, and reduce computational consumption.
- For HPS multiplication, we design a generic module for three variants to maximize operation reuse. We adapt the hybrid key-switching technique for ciphertext relinearization, and fuse tensoring with relinearization in the leveled approach to further reduce the computation of scaling and memory transfer.

Our performance evaluation reveals significant improvements. Compared to OpenFHE running 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 multiplication depth increases, our GPU implementation of the leveled HPS variant consistently outperforms, achieving around 4× the speed of other variants in our implementation, showing a potential in specific applications.

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 $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. We use bold, italic lowercase letters to represent polynomials, e.g. $\boldsymbol{f} := \sum_{i=0}^{N-1} f_i X^i$, which are elements of the ring $\mathcal{R}$. We use $\mathcal{U}_Q$ to indicate the uniform distribution over $\mathbb{Z}_Q$, and $\mathcal{X}_k$ and $\mathcal{X}_e$ are used to denote two distinct probability distributions over the ring $\mathcal{R}$. The notation $\boldsymbol{f} \leftarrow \mathcal{S}$ signifies that the element $\boldsymbol{f}$ is sampled according to the distribution $\mathcal{S}$. The notation $[a]_q$ stands for the integer a modulo q.

B. Residue Number System

The Residue Number System (RNS) is often employed to accelerate the computation of 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 $\mathbb{Z}_Q \simeq \Pi_{i=1}^{L} \mathbb{Z}_{q_i}$, impling a large integer in $\mathbb{Z}_Q$ can be decomposed into residues in $\mathbb{Z}_{q_i}$ and each fits 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 coefficients as multi-precision integers, the RNS representation of $\boldsymbol{x}$ can be expressed as $(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(L)})$, 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. The conversions between the original and the RNS representation, can be expressed as follows:

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

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

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

RNS Base Conversion. The conversion of the RNS base of an element is a necessary step 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 respective RNS bases can be defined as $\mathcal{Q} := \{q_1, q_2, \ldots, q_L\}$ and $\mathcal{B} := \{r_1, r_2, \ldots, r_K\}$. To convert the base of a ring element $\boldsymbol{x} \in \mathcal{R}_Q$ from base $\mathcal{Q}$ to base $\mathcal{B}$, two methods are currently proposed. The first is the Bajard-Eynard-Hasan-Zucca (BEHZ) method [12], represented as:

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

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

The second method is the Halevi-Polyakov-Shoup (HPS) method [13]. Unlike the BEHZ method, the HPS method computes an exact conversion, outputting $[[\boldsymbol{x}]_Q]_P$. This method can be expressed as follows:

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

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 initially computes the intermediate value $\boldsymbol{y}^{(i)} := [\boldsymbol{x}^{(i)} \cdot \tilde{q}_i]_{q_i}$ using integer arithmetic, followed by the computation of $\boldsymbol{z}^{(i)} := \boldsymbol{y}^{(i)}/q_i$ 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 modulus respectively. Below, we describe the commonly used RNS-variant BFV scheme for efficiency, where the modulus Q is represented as the product of a series of coprime integers, i.e., $Q = \Pi_{i=1}^{L} q_i$, and we denote P as the special modulus.

Key Generation. The key generation procedure produces keys used in encryption and evaluation based on the desired security level, including the public key $\mathbf{pk}$, the secret key $\mathbf{sk}$, and the 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 corresponding keys are then defined as follows:

- 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 [24].

Encryption. Given a plaintext $\boldsymbol{m} \in \mathcal{R}_t$, the encryption procedure generates a ciphertext $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1)$, encoding $\boldsymbol{m}$ in the MSD with a scaling $\Delta := \lfloor Q/t \rfloor$. This procedure includes symmetric and asymmetric encryption, differing by the keys used in encryption. 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 as follows:

- 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 two ciphertexts $\mathbf{ct}_1 = (\boldsymbol{c}_{1,0}, \boldsymbol{c}_{1,1}), \mathbf{ct}_2 = (\boldsymbol{c}_{2,0}, \boldsymbol{c}_{2,1}) \in \mathcal{R}_Q^2$ involves two steps:

- Tensoring. First, takes the two ciphertexts $\mathtt{ct}_1$ 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. Second, relinearize the triple to a ciphertext with two ring elements, by computing $(\boldsymbol{c}_0^{\mathrm{r}}, \boldsymbol{c}_1^{\mathrm{r}}) := \lfloor P^{-1} \boldsymbol{c}_2^* \cdot \mathbf{rlk} \rceil$. Then, output $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1) := ([\boldsymbol{c}_0^* + \boldsymbol{c}_0^{\mathrm{r}}]_Q, [\boldsymbol{c}_1^* + \boldsymbol{c}_1^{\mathrm{r}}]_Q)$.

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

D. GPU Fundamentals

GPUs have emerged as powerful tools for high-performance computing, due to their massive parallel processing capabilities. The most basic execution unit within a GPU is the thread, which runs a kernel in parallel with other threads to enhance the overall processing speed. Every thread is equipped with its private local memory and registers. Threads on a GPU 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. Among these, global memory is the largest, but it suffers from comparatively high latency.

GPUs operate with an instruction set that involves various types of operations, including arithmetic, logic operations, and memory access operations. A notable feature is that GPUs typically have 32-bit registers, and thus 64-bit operations are constructed over the 32-bit arithmetic. This design consideration is crucial for understanding and optimizing the performance of GPU-based applications.

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: $\quad \boldsymbol{f}^{(i)} := [\mathtt{Conv}_{Q \to i}^{\mathrm{BEHZ}}([\gamma t \cdot \boldsymbol{x}]_Q) \times [-Q^{-1}]_i]_i$
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** $\boldsymbol{m}^{(t)}$

III. BFV Variants

In this section, we delve into a comprehensive exploration and analysis of various variants of BFV. We begin by focusing on two primary methods, i.e., BEHZ and HPS, that address the issues of simple scaling in decryption and complex scaling in tensoring. Each method comes with its own set of strengths and challenges, and we elucidate these with careful consideration. Furthermore, we examine recent innovative adaptations of these methods, including techniques to expand the input size in HPS and strategies to minimize computational complexity and memory usage in tensoring. Finally, we discuss the process of relinearization, particularly the implementation of hybrid key-switching within the BFV scheme.

A. Simple Scaling in Decryption

The decryption procedure in the BFV scheme operates by first computing $\boldsymbol{x} := [\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}]_Q$ in $\mathcal{R}_Q$. This is then scaled by a factor of t/Q and rounded to the nearest integer to compute $\boldsymbol{m} = [\lfloor t/Q \cdot \boldsymbol{x} \rceil]_t$, where each coefficient of $\boldsymbol{x}$ undergoes the scaling-and-rounding process. Notably, the BEHZ and HPS methods differ in how they implement this operation. The pseudocode of utilizing the two method for decryption are represented in Algorithm 1 and 2 respectively.

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

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

Algorithm 2 $\texttt{Dec}_{\texttt{HPS}}$: 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: $[\boldsymbol{m}]_t$

$\boldsymbol{x} := \langle \mathbf{ct}, \mathbf{sk} \rangle$
for $j \in [0, N)$ **do** ▷ Scaling-and-rounding
  $w := 0, v := 0.0$ ▷ $\boldsymbol{x} := \{\boldsymbol{x}^{(i)}\}$
  for $i \in [1, L]$ **do** ▷ $\boldsymbol{x}^{(i)} := \sum_{j=0}^{N-1} x_j^{(i)} X^j$
    $w := w + [x_j^{(i)} \cdot \omega_i]_{q_i}$ ▷ Integral part
    $v := v + x_j^{(i)} \cdot \theta_i$ ▷ Fractional part
  $sum := w + v$
  $m_j := \lfloor sum - sum/t \rceil$ ▷ Modular reduction
return $\boldsymbol{m} := \sum_{j=0}^{N-1} m_j X^j$

This provides exact integer division that can be performed under the RNS representation. The process of base conversion from $\mathcal{Q}$ to t causes the term $t[\boldsymbol{x}]_Q$ to vanish during computation modulo t. After base conversion, the remaining term $[[t \cdot \boldsymbol{x}]_Q]_t$ can be obtained. Then, an auxiliary modulo γ that is coprime to both t and Q is introduced to correct the error produced by base conversion and the replacement of rounding. For $i \in \{t, \gamma\}$, we have

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

In this equation, 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 enables us to reduce $\gamma([\boldsymbol{m}]_t + t\boldsymbol{\delta}_0)$ modulo t to $\gamma[\boldsymbol{m}]_t$ modulo t. As such, the decryption result $[\boldsymbol{m}]_t$ can be recovered by multiplying $[\gamma^{-1}]_t$ by $\gamma[\boldsymbol{m}]_t$.

HPS Method. The HPS method focuses on each coefficient of the ring elements. To simplify the notation, we use $x \in \mathbb{Z}_Q$ and $m \in \mathbb{Z}_t$ to denote the coefficients. In this case, the RNS representation of x is given by $(x^{(1)}, x^{(2)}, \ldots, x^{(L)})$. Applying equation (2) to decompose x, each coefficient of the decryption result $\boldsymbol{m}$ is computed by the following sequence of operations:

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

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 $\tilde{\theta}_i = \theta_i + \epsilon_i$ instead of the accurate θ_i in practice, with an error term given by $|\epsilon_i| < 2^{-\kappa}$ when κ-bit precision is available. Consequently, the total error term is defined as $\epsilon = \sum_{i=1}^{L} x_i \epsilon_i$. To ensure the correctness of decryption, we require that $||\epsilon||_\infty < 1/4$. However, as the IEEE 754 double-precision format provides 53-bit precision, the HPS method does not accommodate large moduli. This implies a constraint on the magnitude of $L \cdot q_{\max}$, where $q_{\max} = \max_i q_i$, thus introducing a significant limitation that high precision floating-point arithmetic becomes a necessity when the size of the modulus is large.

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

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

In the above equations, $t[\tilde{q}_i \cdot B^j]_{q_i}/q_i := \omega_{i,j} + \theta_{i,j}$, where $\omega_{i,j} \in \mathbb{Z}_t$ and $\theta_{i,j} \in [-1/2, 1/2)$ can be pre-computed. With this methodology, we can enlarge the input size and make the HPS method more flexible in handling large modulus.

Algorithm 3 $\texttt{Tensor}_{\texttt{BEHZ}}$: 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 $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*)$

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 of BFV homomorphic multiplication first computes the tensor product $\mathbf{ct}'_1 \times \mathbf{ct}'_2$ to yield a triple $\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1, \boldsymbol{c}'_2)$. Subsequently, it scales this triple down by t/Q to produce $\mathbf{ct}^* = [\lfloor \frac{t}{Q} \cdot \mathbf{ct}' \rceil]_Q \mathcal{R}_Q^3$. In this context, it is crucial that the product is calculated without any modular reduction prior to scaling. To ensure this correctness, we employ an auxiliary modulus $R := \Pi_{k=1}^{K} r_k > Q$, where each $\boldsymbol{c}_{i,h}$ is extended to modulo QR to facilitate the product calculation. This auxiliary base is denoted as $\mathcal{B} = r_1, \ldots, r_K$. To address the complex scaling involved here, two distinct approaches have been proposed by BEHZ [12] and HPS [13]. We summarize the two different tensoring processes in Algorithm 3 and Algorithm 4.

Algorithm 4 $\texttt{Tensor}_{\texttt{HPS}}$: 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: $\mathbf{ct}^* := (\boldsymbol{c}_0^*, \boldsymbol{c}_1^*, \boldsymbol{c}_2^*) \in \mathcal{R}_Q^3$
1: **for** $i \in \{1,2\}$, $j \in \{0,1\}$ **do**
2: $\quad \boldsymbol{\varsigma}_{i,j} := \texttt{Conv}^{\text{HPS}}_{\mathcal{Q}\to\mathcal{B}}(\boldsymbol{c}_{i,j})$
3: $\quad \boldsymbol{c}'_{i,j} := \boldsymbol{c}_{i,j} || \boldsymbol{\varsigma}'_{i,j}$, $\mathbf{ct}'_i := (\boldsymbol{c}'_{i,0}, \boldsymbol{c}'_{i,1})$ ▷ Concatenate
4: $\mathbf{ct}' := (\boldsymbol{c}'_0, \boldsymbol{c}'_1, \boldsymbol{c}'_2) := \mathbf{ct}'_1 \times \mathbf{ct}'_2$ ▷ $\mathbf{ct}'$ in $\mathcal{Q} \cup \mathcal{B}$
5: **for** $h \in [0,2]$ **do** ▷ Scaling-and-rounding
6: $\quad$ **for** $j \in [0, N)$ **do** ▷ $\boldsymbol{c}'_h := \{\boldsymbol{c}'^{(i)}_h, \boldsymbol{c}'^{(k)}_h\}$
7: $\quad\quad w := 0, v := 0.0$
8: $\quad\quad$ **for** $i \in [1, L]$ **do** ▷ $\boldsymbol{c}'^{(i)}_h := \sum_{j=0}^{N-1} c'^{(i)}_{h,j} x^j$
9: $\quad\quad\quad v := v + c'^{(i)}_{h,j} \cdot \theta_i$
10: $\quad\quad v := \lfloor v \rceil$ ▷ Fractional part
11: $\quad\quad$ **for** $k \in [1, K]$ **do**
12: $\quad\quad\quad$ **for** $i \in [1, L]$ **do**
13: $\quad\quad\quad\quad w := w + [c'^{(i)}_{h,j} \cdot \omega_{i,k}]_{r_k}$
14: $\quad\quad\quad w := w + [c'^{(k)}_{h,j} \cdot \lambda_k]_{r_k}$ ▷ Integral part
15: $\quad\quad\quad \tilde{c}^{(k)}_{h,j} := [w + v]_{r_k}$ ▷ $\tilde{\boldsymbol{c}}^{(k)}_h := \sum_{j=0}^{N-1} \tilde{c}^{(k)}_{h,j} x^j$
16: $\quad \boldsymbol{c}^*_h := \texttt{Conv}^{\text{HPS}}_{\mathcal{B}\to\mathcal{Q}}(\tilde{\boldsymbol{c}}_h)$ ▷ $\tilde{\boldsymbol{c}}_h := \{\tilde{\boldsymbol{c}}^{(k)}_h\}$
17: **return** $\mathbf{ct}^* := (\boldsymbol{c}^*_0, \boldsymbol{c}^*_1, \boldsymbol{c}^*_2)$

BEHZ Method. Due to the fact that the conversion $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}\to\mathcal{B}}(\boldsymbol{c}_{i,h})$ can introduce a Q-overflow, symbolized as $\boldsymbol{c}_{i,h} + Q\boldsymbol{u}$, an extra modulus $\tilde{m}$ is introduced that allows us to deploy the Small Montgomery Reduction technique to mitigate this overflow. Initially, the ciphertexts are multiplied by $\tilde{m}$, and then converted to bases $\mathcal{B}_{sk}$ and $\{\tilde{m}\}$ respectively, where in the first conversion we achieve $\boldsymbol{\varsigma}_{i,h} = [\tilde{m}\boldsymbol{c}_{i,h}]_Q + Q\boldsymbol{u}_{i,h}$. Next, the Small Montgomery Reduction method 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 earlier γ-correction method is insufficient for exact rounding in this complex scaling scenario. Hence, we apply the fast RNS flooring method for approximating the division of Q from $\boldsymbol{c}$ under modulo i as: $[(\boldsymbol{c} - \texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}\to i}([\boldsymbol{c}]_Q)) \times [Q^{-1}]_i]_i$. Consequently, for each $\boldsymbol{c}'_h$, $h \in [0,2]$, the following equation holds true in base $\mathcal{B}_{sk}$:

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

where $||\boldsymbol{u}'_h||_\infty \leq L$.

Finally, we employ another extra modulus m_{sk} to perform an exact conversion back to the original base:

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

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

HPS Method. The HPS variant [13] completes the computation in two stages. Initially, the scaling process is applied as in (5) to compute $\tilde{\boldsymbol{c}}_h = [\lfloor \frac{tR}{QR} \cdot \boldsymbol{c}'_h \rceil]_R$, $h \in [0,2]$. This equates to executing scaling over the modulus tR and then discarding the RNS component of modulus t, which yields:

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

Here, $\tilde{Q}_i := [(q^*_i R)^{-1}]_{q_i}$, $\tilde{Q}_k := [(Qr^*_k)^{-1}]_{r_k}$, and $r^*_k = R/r_k$. Subsequently, $\tilde{\boldsymbol{c}}_h$ is achieved under base $\mathcal{B}$, which are then converted to obtain $\boldsymbol{c}^*_h \in \mathcal{R}_Q$. Here, $W_i + \theta'_i$ is the value for $\frac{t\tilde{Q}_i R}{q_i}$, with $W_i \in \mathbb{Z}_R$ representing the integral part and $\theta'_i \in [-\frac{1}{2}, \frac{1}{2})$ the fractional part. To account for the integral part, 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. In order to ensure accurate tensoring, the auxiliary modulus R should be slightly larger than Q. In practice, this is often achieved by setting $K = L+1$. However, a more compact method proposed in [14] suggests switching the modulus of one of the two ciphertexts from Q to R. Consequently, after multiplying the two ciphertexts, the noise magnitude transitions from a multiple of Q^2 to a multiple of QR that diminishes modulo QR. This approach provides an advantage by allowing the choice of $R \approx Q$, which leads to a more compact parameter configuration where $K = L$, thereby reducing computational complexity. The distinctions from Algorithm 4 are as follows:

- Firstly, in line 2, the RNS base of one of the ciphertexts, for example, $\mathbf{ct}_1$, is switched from base $\mathcal{Q}$ to $\mathcal{B}$, and then extended to base $\mathcal{Q} \cup \mathcal{B}$.
- Secondly, during the scaling-and-rounding process, instead of dividing by Q to obtain residues in base $\mathcal{B}$ and then switching to $\mathcal{Q}$, division by P is performed to directly get residues in base $\mathcal{Q}$.

As a result, this method reduces the total number of base conversions by 1, and the reduced size of K decreases the complexity of each base conversion calculation.

Leveled HPS. Taking inspiration from the leveled structure of the BGV scheme [3], a leveled approach can also be applied to BFV multiplication [14] by switching to $Q_l := \Pi_{i=1}^{l} q_i$ for performing multiplication, thereby reducing both computational and memory complexity of BFV multiplication. The process is as follows:

- Scale the ciphertexts by $\frac{Q_l}{Q}$ to obtain $\widetilde{\mathbf{ct}}_1 := \lfloor \frac{Q_l}{Q}\mathbf{ct}_1 \rceil, \widetilde{\mathbf{ct}}_2 := \lfloor \frac{Q_l}{Q}\mathbf{ct}_2 \rceil \in \mathcal{R}^2_{Q_l}$.
- Extend the RNS base to acquire $\mathbf{ct}'_1$ and $\mathbf{ct}'_2 \in \mathcal{R}^2_{Q_l R_l}$, 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 $\mathbf{ct}' := \mathbf{ct}'_1 \times \mathbf{ct}'_2 := (\boldsymbol{c}'_0, \boldsymbol{c}'_1, \boldsymbol{c}'_2) \in \mathcal{R}^3_{Q_l R_l}$.
- 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 $\widetilde{\mathbf{ct}} := (\tilde{\boldsymbol{c}}_0, \tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) \in \mathcal{R}^3_{Q_l}$.
- Lastly, scale $\widetilde{\mathbf{ct}} \in \mathcal{R}^3_{Q_l}$ up to the original Q by $\mathbf{ct} := \lfloor \frac{Q}{Q_l}\widetilde{\mathbf{ct}} \rceil$ to get the result $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}^3_Q$.

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: $\mathbf{ct} := (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_Q$

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

Compared to the compact HPS variant that reduces the size of r_k by 1, the leveled approach reduces both the magnitude of the inner loop from L or K to l, at the cost of more base conversions. However, these conversions are more efficient since the size of the RNS base of the input and output is diminished. The selection criteria for l is provided in [25]. 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 $\frac{Q}{Q_l} = \Pi_{i=l+1}^{L} q_i$. In detail, this entails 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, as initially proposed for the CKKS scheme [26], is notable for its ability to handle real number input, but it can also be adapted for homomorphic encryption schemes over finite fields such as BFV. In our adaptation, we select the special modulus as $P = \Pi_{i=1}^{\ell} p_i$ and define the decomposition number as $\mathtt{dnum} := \lceil L/\ell \rceil$. 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 hybrid key-switching technique in Algorithm 5. The relinearization key, $\mathbf{rlk}$, consists of a set of tuples: $\mathbf{rlk} := \{([-\boldsymbol{a}_j \boldsymbol{s} + \boldsymbol{e}_j + PT_j \boldsymbol{s}^2]_{PQ}, \boldsymbol{a}_j)\}_{j \in [0, \mathtt{dnum})}$, wherein the decomposition base, T_j, is set as Q_j^*.

Given the scale-invariance of the BFV scheme, the base extension of $\boldsymbol{c}_2^*$ can be eliminated as per the original methodology [26]. The decomposition process is simplified by extracting the appropriate RNS residues and subsequently multiplying them with $\tilde{Q}_j$. In the next phase, the RNS base is extended from $\mathcal{D}_j$ to $\mathcal{Q} \cup \mathcal{P}$, and the inner-product with $\mathbf{rlk}$ is computed to yield the accumulation result, $(\boldsymbol{t}_0, \boldsymbol{t}_1) \in \mathcal{R}_{QP}^2$. The resulting 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.

In conjunction with this, the [13] can be efficaciously implemented in this context to improve computational efficiency. The optimization is achieved by designating the decomposition base T_j to be the product $\tilde{Q}_j Q_j^*$ This adjustment obviates the necessity of scalar multiplication with $\tilde{Q}_j$ in the decomposition stage, thereby streamlining the process.

TABLE I
BREAKDOWN OF OPERATIONS FOR THE BEHZ AND HPS VARIANTS OF BFV

Operations	Break down
$\mathtt{Dec}_{\mathrm{BEHZ}}$	Inner product, base conversion, subtraction, scalar multiplication
$\mathtt{Dec}_{\mathrm{HPS}}$	Inner product, simple scaling-and-rounding
$\mathtt{Tensor}_{\mathrm{BEHZ}}$	Scalar multiplication, base conversion, tensor product, scalar multiplication with accumulation / subtraction
$\mathtt{Tensor}_{\mathrm{HPS}}$	Base conversion, complex scaling-and-rounding
`Relin`	Base conversion, inner product, addition

IV. FRAMEWORK AND OPTIMIZED IMPLEMENTATION

A. Hierarchical Operation Breakdown

Based on the comprehensive analysis undertaken earlier, we conduct a hierarchical operation breakdown, deconstructing complex operations into a series of reusable RNS level polynomial operations, as depicted in Table I. In our approach, the HPS variant comprises operations include the product and a specific form of scaling-and-rounding. This stands in contrast to the BEHZ variant, which necessitates a more intricate sequence of operations, including product, multiple base conversions, and various other arithmetic operations. Consequently, our implementation follows a three-tiered structure that supports diverse computations.

- **Low-level Ring Arithmetics:** This foundational layer sets the groundwork for our implementation, commencing with our refined integer modular operations with the least instructions. Based on this, we further implement polynomial modular operations to extend the capability.
- **RNS-level Operations:** In this layer, we perform arithmetic operations over polynomials that are represented under the RNS. Capitalizing on the inherent parallelism of GPU architecture, we batch execute these operations, thereby enhancing performance. We aim to implement the operations in a generic manner, ensuring their wide applicability and reusability across different cases.
- **High-level Module of Homomorphic Operations:** The final tier concentrates on developing generic modules for different variant operations, leveraging previously implemented reusable operations. We explore finely-tuned fusion approaches to reduce memory access and usage, thereby boosting the performance.

This layered implementation strategy fosters operational efficiency and enhances the versatility. It optimizes the execution of both BEHZ and HPS variants and provides room for potential expansion to cover more computational use cases.

B. Ring Arithmetic

Below, we delve into the implementation details of our modular operations and polynomial multiplication, which form the backbone of our framework.

Modular Reduction. For modular reduction, we use the Barrett reduction method, which replaces costly division with quicker multiplication and bit-shifting. This approach necessitates a pre-computation step where $\mu := \lfloor \frac{2^\beta}{q} \rfloor$, with β being set as the machine word size. Subsequently, the reduction

Algorithm 6 Modular Multiplication

function `ModRed`(a, q, μ) ▷ Reduction of 64-bit integer
  $tmp := \_\_\texttt{umul64hi}(a, \mu)$ ▷ $\mu := \lfloor \frac{2^{64}}{q} \rfloor$
  return $a - tmp \cdot q$
function `CModMul`(a, ζ, q, μ) ▷ Constant multiplication
  $tmp := \_\_\texttt{umul64hi}(a, \mu)$ ▷ $\mu := \lfloor \frac{\zeta \cdot 2^{64}}{q} \rfloor$
  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

result is 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 and maximize efficiency, 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. Based on this, we employ the Shoup technique [27] for quicker modular multiplication when one input is a constant, denoted as `CModMul`. When multiplying a and a constant ζ, it can be merged into pre-computation by setting $\mu := \lfloor \frac{\zeta \cdot 2^{64}}{q} \rfloor$. This saves one multiplication instruction, bringing the cost of modular multiplication to the same as reduction. For general cases with two variable inputs, we adopt the optimized modular reduction from [20], and develop a modular multiplication `ModMul` for two 64-bit integers. As CUDA does not support 128-bit registers, two 64-bit multiplication instructions are utilized to obtain the 128-bit multiplication 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. Polynomial multiplication poses a significant performance challenge, which we address by employing the Number Theoretic Transform (NTT). For polynomials $\boldsymbol{f}, \boldsymbol{g} \in \mathcal{R}_q$, the forward negacyclic NTT is formulated as $\hat{f} := \mathsf{NTT}(f)$, $\hat{f}_j = \sum_{i=0}^{n-1} f_i \zeta^{(2i+1)j} \pmod q$, and the inverse is $f := \mathsf{INTT}(\hat{f})$, $f_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{f}_j \zeta^{-(2i+1)j} \pmod q$, where ζ is the primitive $2n$-th root of unity. This transformation allows us to perform multiplication as $fg := \mathsf{INTT}(\mathsf{NTT}(f) \cdot \mathsf{NTT}(g))$, reducing computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. We adopt the commonly used hierarchical NTT implementation [28], [29], and partition the process into two kernels, each responsible for different transformation levels. For every forward or inverse NTT operation involving N elements, we use 8 per-thread implementation, where each thread in a kernel loads eight residues into the registers and perform a radix-8 transformation. SMEM is used for temporary output storage between each radix-8 transformation to minimize GMEM interaction.

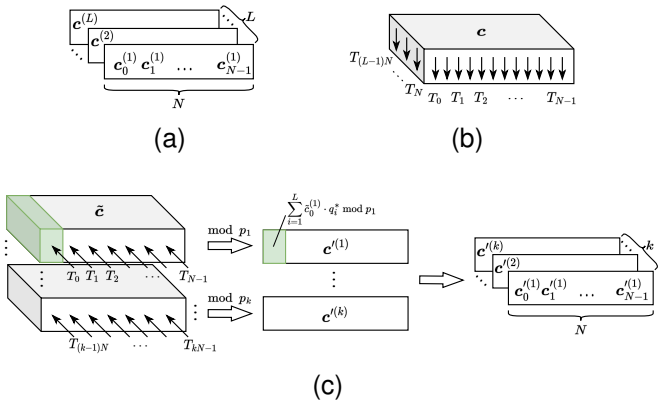

Fig. 1. The structure of our base conversion implementation prior to the loop unroll optimization. (a) A polynomial in RNS representation. (b) The first kernel of the base conversion processing at an $I \cdot N$ dimension. (c) The second kernel of the base conversion working at an $O \cdot N$ dimension.

C. Base Conversion

Base conversion for a polynomial $\boldsymbol{c}$ in RNS representation is a critical operation. 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 [20], could lead to inefficient utilization of GPU hardware resource. Conversely, focusing on a single dimension (e.g., $O \cdot N$) burdens each 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. 1b. 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. 1c. 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 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. Memory constraints due to excessive loading and storing operations can affect the performance of the two kernels, particularly in instances where constants are repeatedly loaded by each thread. To mitigate this, we apply loop unrolling, as demonstrated in Algorithm 7. We establish an unroll factor, F, wherein each thread processes F coefficients of I residues. This not only reduces the loads of pre-computed values (q_i^*, r_k, μ_k) from F to one, but also enables the substitution of 64-bit operations with wider instructions, hence reducing pipeline overhead. Note that in Algorithm 7, we use `WideLoad` and `WideStore` to denote general case wide load and store instructions, such as

Algorithm 7 Unrolled base convert acc

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

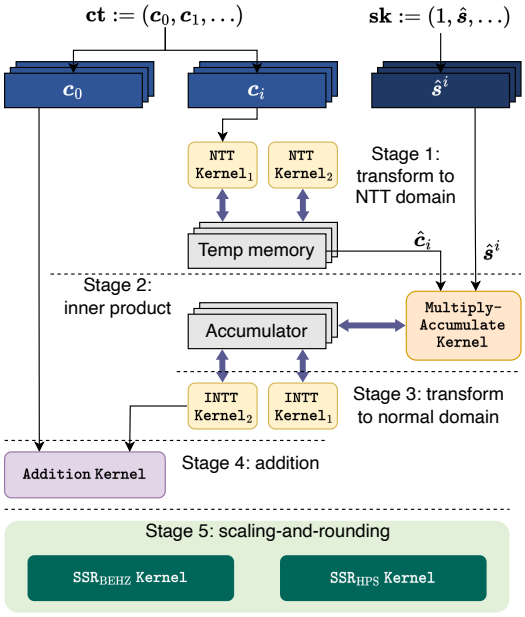

Fig. 2. The architecture of generic decryption module designed to accommodate both BEHZ and HPS techniques.

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. 2. The decryption in the BFV scheme first compute the inner product of $\mathbf{sk}$ and $\mathbf{ct}$, followed by scaling the result by $\frac{t}{Q}$ to the nearest integer. The scaling-and-rounding phase is the primary differentiation point between the BEHZ and HPS, with the former utilizing modular arithmetic, and the latter harnessing precision.

In the first step, the inner product of $\mathbf{sk}$ and $\mathbf{ct}$ is computed. It is worth noting that relinearization is not a compulsory procedure, which means $\mathbf{ct}$ 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

Algorithm 8 SSR_{BEHZ}: BEHZ simple scaling-and-rounding

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

Algorithm 9 SSR_{HPS}:HPS simple scaling-and-rounding

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

the NTT kernels to transition elements in $\mathbf{ct}$, barring $\boldsymbol{c}_0$, to the NTT domain, and subsequently carry out efficient polynomial multiplication with $\boldsymbol{s}^i$. To minimize memory overhead, we establish a buffer in GMEM for memory reuse. Following this, the accumulated result is converted back added to $\boldsymbol{c}_0$.

The subsequent scaling-and-rounding phase is contingent on the specific method used. Our implementations of the BEHZ and HPS methods, denoted as SSR_{BEHZ} and SSR_{HPS}, respectively, are outlined in Algorithm 8 and 9.

Optimizations to BEHZ Method. In contrast to the HPS method, which requires only coefficient-wise addition and multiplication, the BEHZ method is more complex, involving four distinct stages, and offering two different levels of parallelism corresponding to the dimensionality of the operands. For the first two kernels, which conduct modular multiplication with γt and the first kernel of base conversion `Conv_ScalarMul`, are essentially scalar modular multiplication. Therefore, we fuse them into a single kernel with $L \cdot N/F$ threads, thus eliminating writing and reading $\boldsymbol{x}'$ from the GMEM. Following this, we launch the second kernel of base conversion, `Conv_MatMul`, which utilizes $\lceil K \cdot N/F \rceil$ threads to conduct the unrolled modular multiplication with the matrix. Then, we adjust $\boldsymbol{f}^{(r)}$ to the centered remainder of $\boldsymbol{f}$ modulo γ, compute $\gamma - \boldsymbol{f}^{(r)}$ to derive the reflexivity, add this to $\boldsymbol{f}^{(t)}$ to obtain $\boldsymbol{f}^{(t)} - \boldsymbol{f}^{(r)}$, and then multiply by $[\gamma^{-1}]_t$ to produce the final $\boldsymbol{m}$. This process is optimized by fusing to one kernel.

E. Homomorphic Multiplication

Homomorphic multiplication in the BFV scheme poses a significant computational challenge. We aim at accelerating

all BEHZ and HPS variants of this operation. These variants come with their unique sets of merits and demerits, making it essential to focus on all in order to devise more comprehensive and efficient solutions.

The existing researches predominantly concentrate on enhancing the speed of the BEHZ variant. For instance, the study documented in [19] focuses on small parameter sets and employs 10 kernels, leaving ample room for potential computational and memory optimizations. The research outlined in [20] focuses on compact designs and divide the entire procedure into three processes to facilitate kernel fusing. They do provide valuable insights, but there remain several underexplored optimizations within the context of BEHZ-type multiplication. On the other hand, the HPS variant has received relatively less attention in the literature. The work [21] confines the data type to 32-bit arithmetic, thereby restricting the scope and applicability of their solutions.

In contrast, our design strategy primarily aims to accommodate a more generalized case, and addresses these limitations. We propose a series of novel enhancements aimed at improving both schemes. Below, we elucidate 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:** Noticing that the first phase of base conversion, `Conv_Scalar`, comprises scalar multiplication computation and that the second phase is frequently followed by arithmetic operations such as scalar multiplication and subtraction in the same bases, we have investigated several fusion strategies for performance enhancement. We adapted multiplication with other constants to integrate into the first phase. By incorporating $\tilde{m}$ into the pre-computation of the conversion matrix, we can directly derive $[\tilde{m}\boldsymbol{c}]_Q$, thus eliminating a modular multiplication for each coefficient with $\tilde{m}$. Furthermore, we have fused multiply-and-subtract computation into the second phase of base conversions in both the fast flooring and `ConvSK` stages, given that these computations are conducted under identical RNS bases. These optimizations have significantly minimized time-consuming GMEM access of the polynomials.
- **Inter-Conversion Fusion:** In BEHZ-type tensoring computation, it is common to convert one polynomial to two different bases successively. Conventionally, this would involve calling two base conversion functions. However, we noted that the initial phase of these conversions is the same and can be reused, reducing computation. Specifically, in the process of reducing Q-overflow prior to the tensor product, we observed that two base conversions, i.e., $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}\rightarrow\mathcal{B}_{sk}}$ and $\texttt{Conv}^{\text{BEHZ}}_{\mathcal{Q}\rightarrow\{\tilde{m}\}}$, converting $[\tilde{m}\boldsymbol{c}_{i,h}]_Q$ to base $\mathcal{B}_{sk}$ and $\{\tilde{m}\}$ respectively, share the same `Conv_Scalar`process. We fused these two operations, thus saving one stage of the conversions. A similar approach was adopted in `ConvSK`, where $[\tilde{\boldsymbol{c}}_h]_{\mathcal{B}}$ are converted to base $\{m_{sk}\}$ and base $\mathcal{B}$, respectively.

2) Generic HPS-type Tensoring Design: We have developed a generic HPS-type tensoring framework which is compatible with all three HPS variants. Given the two input ciphertexts $\mathbf{ct}_1, \mathbf{ct}_2 \in \mathcal{R}_Q$, the framework extends the base of the first ciphertext $\mathbf{ct}_1$ to $\mathcal{Q}\cup\mathcal{B}$, or scales it down to $\mathcal{Q}_l$ before extending to $\mathcal{Q}_l\cup\mathcal{B}_l$. The second ciphertext $\mathbf{ct}_2$ is manipulated to extend its base to $\mathcal{Q}\cup\mathcal{B}$, convert to $\mathcal{B}$, then extend to $\mathcal{Q}\cup\mathcal{B}$, or convert to $\mathcal{B}_l$ and then extend to $\mathcal{Q}_l \cup \mathcal{B}_l$ according to the three variants. 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.

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

Input: $\boldsymbol{c} := \{\boldsymbol{c}^{(i)}\},\ i \in [0, I)$
Output: $\boldsymbol{c}' := \{\boldsymbol{c}'^{(i)}\},\ i \in [0, O)$

```
tid ∈ [0, N)
sum_I := {0, 0}, sum_F := 0.0
for k ∈ [O, I) do
    sum_F := sum_F + c[kN + tid] · θ̃_{k−O}
for k ∈ [0, O) do
    for j ∈ [O, I) do
        sum_I := sum_I + c[jN + tid] · ω_{k,j−O}
    sum_I := sum_I + c[kN + tid] · λ_k
    c'[kN + tid] = ModRed(ModRed(sum_I) + ⌊sum_F⌉)
return c'
```

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, is defined to handle cases where no overflow occurs. This versatile approach significantly enhances computational efficiency and is a noteworthy contribution towards the advancement of homomorphic encryption.

3) Reducing Computation in Leveled Approach: In the leveled variant, 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 optimized strategy that fuses 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. Moreover, this approach also manages to eliminate the noise that is typically introduced by these two scaling operations. This innovative strategy offers a significant enhancement in computational efficiency and precision, reinforcing the efficacy of the leveled approach in homomorphic encryption.

F. Complexity Analysis

Below, we perform an analytical examination of the BFV variant implementations that are built on our proposed framework. We begin by detailing the pre-computations required for

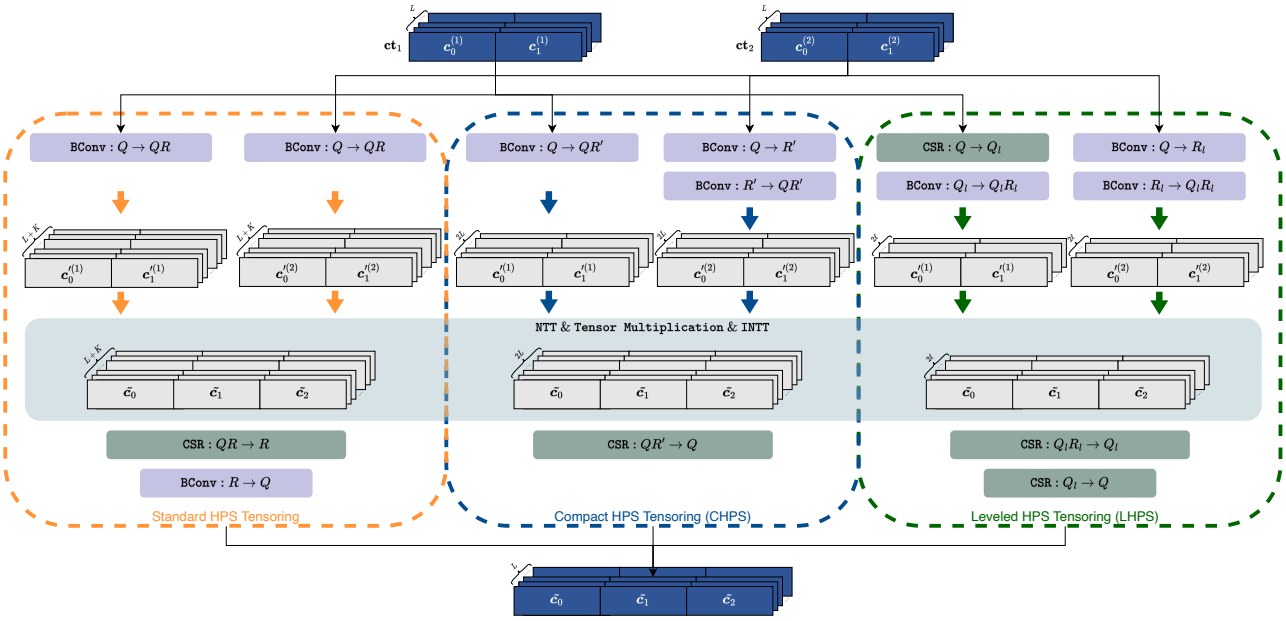

Fig. 3. A unified module for original HPS, compact HPS, and leveled HPS variants.

TABLE II
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	$[\gamma^{-1}]_t$ $[t \cdot \gamma]_{q_i}$ $[-Q^{-1}]_{\{t,\gamma\}}$ $\mathsf{Conv}^{\mathsf{BEHZ}}_{\mathcal{Q}\to\{t,\gamma\}}$	$\gamma^{-1} \bmod t$ $t \cdot \gamma \bmod q_i$ $-Q^{-1} \bmod \{t,\gamma\}$ 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 $[t]_{\mathcal{B}_{sk}}$, $[Q]_{\mathcal{B}_{sk}}$, $[Q^{-1}]_{\mathcal{B}_{sk}}$, $[\tilde{m}^{-1}]_{\mathcal{B}_{sk}}$ $[R]_{q_i}$ $[R^{-1}]_{m_{sk}}$, $[-Q^{-1}]_{\tilde{m}}$ $\mathsf{Conv}^{\mathsf{BEHZ}}$	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 $\mathsf{Conv}^{\mathsf{HPS}}$	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

these variants, and subsequently discuss their computational complexity.

1) Pre-computation: A vital aspect is the pre-computation of certain values that expedite subsequent calculations. For every single modulus q, several pre-computations are undertaken, which we denote as $\mathsf{q_prm}$. Specifically, we compute the value $\mu := \lfloor \frac{2^{64}}{q} \rfloor$ for fast modular reduction, and two sequences of ζ^i and $(\zeta^{-1})^i$ function 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.

In the case of each RNS 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 $k \in [1, K]$. This is executed for both the BEHZ and HPS methods. For the HPS method, additional computations of $[v \cdot Q]_{r_k}$ are needed where $v \in [0, L]$. The total computational cost amounts to $L(K+1)$ for the BEHZ method and $2L + LK + 1$ for the HPS method.

Table II summarizes the pre-computation values for all BEHZ and HPS variants that are implemented in our framework, including the decryption and multiplication operations.

2) Computational Complexity: To provide an overarching view of the computational complexity of our approach, we present a comprehensive summary in Table III. Prior studies such as [14] have performed operation-level analyses, which involved counting the number of operations, including base conversion and NTT, to assess computational overhead. Another work by [21] presented a comparative analysis of BEHZ

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

TABLE IV
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	
$\texttt{HMult}_{\texttt{BEHZ}}$	0.311	0.463	0.398	1.378	0.921	0.486	0.634	0.573	1.426	1.017	20.200	**31.9**×
$\texttt{HMult}_{\texttt{HPS}}$	0.207	0.389	0.289	1.445	0.968	0.310	0.466	0.395	1.293	0.872	14.400	**30.9**×
$\texttt{HMult}_{\texttt{CHPS}}$	0.189	0.323	0.261	1.302	0.836	0.292	0.445	0.367	1.207	0.794	11.800	**26.5**×
$\texttt{HMult}_{\texttt{LHPS}}$	-	0.290	0.255	1.187	0.775	-	0.408	0.318	1.145	0.735	11.400	**27.9**×
$\texttt{Dec}_{\texttt{BEHZ}}$	0.039	0.049	0.046	0.104	0.086	0.066	0.072	0.071	0.108	0.100	1.030	**14.3**×
$\texttt{Dec}_{\texttt{HPS}}$	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 V
PERFORMANCE COMPARISON OF OUR OPTIMIZED BEHZ VARIANT GPU IMPLEMENTATION WITH RELATED WORKS [19]–[21] (MEASURE IN MILLISECONDS). SPEEDUP RATIOS ARE DETERMINED BY COMPARING PERFORMANCE WITH EACH RESPECTIVE WORK ON THE SAME HARDWARE PLATFORM.

Operations	Our work							[19]		[20]	[21]
	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
$\texttt{Tensor}_{\texttt{BEHZ}}$	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%**	-	
$\texttt{Dec}_{\texttt{BEHZ}}$	0.040	0.054	0.131	0.067	0.075	0.129	0.317	-		-	-

and HPS variants, although it was confined to the number of modular operations and floating-point operations.

Our analysis diverges from these previous approaches by providing a more granular evaluation that aligns more closely with the GPU platform. Given that GPU-based implementations of HE tend to be memory-bound, our assessment takes into account the memory access complexity of each variant, including GMEM load, GMEM store, and SMEM access. Furthermore, acknowledging that the number of instructions used by modulo reduction varies depending on data sizes, we offer in-depth count of integer multiplication instructions (IM) and floating-point instructions (FP).

V. PERFORMANCE EVALUATION

In this section, we present the performance of our implementation, and the comparison with related works. Our experimental setup comprises of a variety of computational units. We compile the C/C++ code using g++ 12.2.0 and the GPU implementations with CUDA 11.8 on an Arch Linux system 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 measure the performance of the various procedures in terms of their execution time, which is reported in milliseconds (ms). Note that the reported times also account for the latency associated with data transfer between the CPU and GPU.

A. Performance of BFV Variants

Table IV elucidates the performance outcomes of our tailored BFV variant implementations across two distinct GPUs, with metrics furnished in milliseconds. The `HMult` operation, indicative of homomorphic multiplication, comprises both tensoring and relinearization. Meanwhile, the `Dec` operation corresponds to the decryption procedure.

In the leveled variant, $\texttt{HMult}_{\texttt{LHPS}}$, we present the performance resulting from a single-level drop, except for $N = 2^{13}$

parameter sets where no level decrement is permissible. For comparison, we run the open-source library OpenFHE [7] on our multi-threaded CPU platform to establish a CPU baseline.

Our implementations demonstrate significant speedups compared to the OpenFHE implementation, with the speedup factors range from 14.3× to 31.9×. For the CHPS and LHPS variants, the performances range from 0.189ms to 1.302ms and 0.290ms to 1.187ms respectively on the 3090 Ti GPU, 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.

B. 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 [19]–[21]. The work in [19] represents the most recent and advanced implementation of the BEHZ variant on a GPU. Meanwhile, [21] 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 V. As the work of [19] 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.

C. Sensitivity to Multiplication Depth in Application

To explore the impacts of different variants of homomorphic multiplication in a real-world application scenario, we implemented an application performing recursive homomorphic multiplication in a binary tree pattern. We set the parameters as $logN = 15$, $logQP = 840$, with the number of special moduli being 1.

For BEHZ, HPS, and CHPS variants, the performance remained essentially constant regardless of the increase in multiplication depth. However, the LHPS variant demonstrated a different pattern, with the number of layers dropped continuously increasing with the multiplication depth, leading to a gradual decline in performance.

Nevertheless, at a multiplication depth of 20, the LHPS variant managed to enhance performance by around 4× when compared to the other three 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.

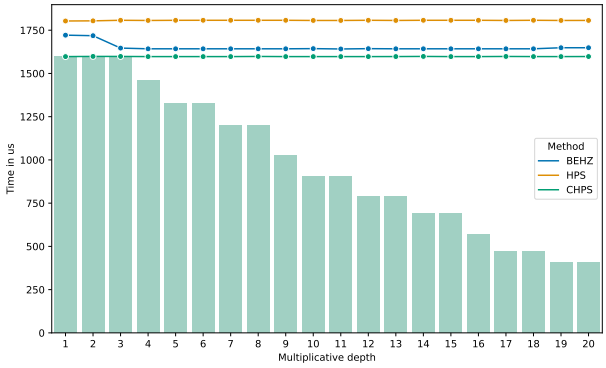

Fig. 4. Performance comparison of four homomorphic multiplication variants within a recursive binary tree application scenario, assessed across varying multiplication depths.

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.

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] 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.

[16] 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.

[17] 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.

[18] 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.

[19] A. S. Özcan, C. Ayduman, E. R. Türkoglu, and E. Savas, "Homomorphic encryption on gpu," IACR Cryptol. ePrint Arch., vol. 2022, p. 1222, 2022.

[20] 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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," IACR Cryptol. ePrint Arch., p. 204, 2021. [Online]. Available: https://eprint.iacr.org/2021/204

[26] 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.

[27] V. Shoup, "NTL: A library for doing number theory," https://libntl.org, 2001.

[28] 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.

[29] 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.

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. He is currently working as a Senior Researcher with Zhejiang Lab, Hangzhou, China. 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.