GPU Implementations of Three Different Key-Switching Methods for Homomorphic Encryption Schemes

Ali Şah Özcan*
alisah@sabanciuniv.edu

Erkay Savaş
erkays@sabanciuniv.edu

November 2024

Abstract

In this work, we report on the latest GPU implementations of the three well-known methods for the key switching operation, which is critical for Fully Homomorphic Encryption (FHE). Additionally, for the first time in the literature, we provide implementations of all three methods in GPU for leveled CKKS schemes. To ensure a fair comparison, we employ the most recent GPU implementation of the number-theoretic transform (NTT), which is the most time-consuming operation in key switching, and evaluate the performance across two fully homomorphic schemes: BFV and CKKS. Furthermore, we highlight the advantages and shortcomings of the three methods in the context of leveled HE schemes, and discuss other aspects such as memory requirements. Our GPU implementation is integrated with HEonGPU Library and delivers up to a ×380 improvement in execution time compared to the Microsoft SEAL Library. Since key switching is a specialized form of the external product common in many HE schemes, our results are directly relevant to time-intensive homomorphic operations such as relinearization and rotation. As homomorphic rotation is one of the most dominant operations in bootstrapping, our results are also applicable in bootstrapping algorithms of BFV, BGV and CKKS schemes.

Keywords: Homomorphic Encryption, GPU Acceleration, External Product, HEonGPU

1 Introduction

Homomorphic Encryption (HE) is a family of encryption schemes, which facilitate performing complex computations over encrypted data without using the secret key. HE is one of the most powerful privacy enhancing technology, which finds numerous practical applications from privacy-preserving machine learning to private information retrieval to private set intersection among many others. Especially, in finance and medical domain, where data privacy is of utmost concern and therefore highly regulated, HE can play an enabler role for the benevolent processing of private data.

Until 2009, there existed so called partially homomorphic encryption schemes such as Paillier [1], Damgard-Jurik [2] and ElGamal [3] that provided support for only one homomorphic operation: either addition or multiplication.Gentry's 2009 work [4] introduced the first fully operational homomorphic encryption (FHE) scheme based on ideal lattices. The key innovation that truly enabled FHE, however, was not just the ability to perform both addition and multiplication (as in somewhat homomorphic encryption schemes, SHE), but the use of bootstrapping to refresh ciphertexts and allow for unlimited operations over ciphertext. Following Gentry's groundbreaking work, more practical HE schemes were developed, particularly those based on the Learning with Errors (LWE) problem and its ring variant (RLWE) [5]. Some of these are classified as somewhat homomorphic encryption (SWHE) schemes, which can only evaluate circuits of limited depth.

In RLWE-based HE schemes, introducing a small amount of noise during public key generation or plaintext encryption is essential for security. As homomorphic computations are performed, this noise gradually increases, resulting in ciphertexts that contain errors. It is critical to control the noise level to ensure that it remains small enough to allow for accurate decryption and recovery of the original plaintext. Modern advanced HE schemes such as BFV [6] [7], BGV [8], CKKS [9], FHEW/TFHE [10,11],

*Developer and corresponding author

and GSW [12] have adopted common strategies to address the noise increase resulting from nonlinear homomorphic encryption operations. These schemes employ decomposition technique that converts a ciphertext component into a vector of small entries before multiplying it with a public key, which is used during homomorphic evaluation operations. This approach effectively controls the noise growth, maintaining computation accuracy and preserving efficiency of homomorphic operations.

Over time, many HE schemes have undergone continuous optimization, with the most notable improvement being the utilization of the residue number system (RNS), which leads to RNS variant of these schemes [13, 14]. This shift has replaced multi-precision modular arithmetic with more efficient, hardware-friendly multi-modulus arithmetic, greatly improving both efficiency and performance. Several CPU-based open-source software libraries, such as Concrete [15], HEAAN [16], HElib [17], Lattigo [18], Microsoft SEAL [19], OpenFHE [20] and PALISADE [21], implement HE schemes, incorporating various optimizations to enhance their practicality and efficiency. Despite substantial advancements, deploying current HE implementations in large-scale practical applications, such as cloud computing, remains a daunting challenge. This is primarily due to the high computational demands and excessive memory and IO requirements of involved mathematical operations in HE schemes. While there have been improvements in algorithms and theoretical foundation pertaining to HE schemes, a performance gap still exists between HE capabilities and the needs of real-world applications. A promising solution to this issue is the use of hardware accelerators. By utilizing parallel architectures such as GPU [22–25], FPGA [26], and ASIC [27], we can achieve significant performance enhancements, making HE implementations more practical and efficient.

The external product, a commonly employed operation by many HE schemes for key switching, rotation and relinearization, is defined as the function $\mathcal{R}_Q \times \mathcal{R}_{\tilde{Q}}^{\ell} \rightarrow \mathcal{R}_{\tilde{Q}}$, where $\mathcal{R}_Q$ is a polynomial ring and $\mathcal{R}_{\tilde{Q}}^{\ell}$ stands for vector of polynomials of size ℓ. Simply put, it consists in high number of multiplications of very high degree polynomials with large coefficients. In this work, we focus on the the key switching operations (relinearization, rotation, and key switching) for the CKKS and BFV schemes, and provide state-of-the-art GPU implementations for three efficient methods for key switching in the literature. They will be referred to as Method I [28, 29], Method II [28, 29], and Method III [30] in this paper. It is important to note that while Method I and Method II are often referred to as hybrid key switching [29], we treat them as distinct methods in this work due to implementation differences.

Our implementation of the three methods are accessible in the open-source GPU library[1] "HEonGPU"-[31], which consists in high performance implementations of all operations of the CKKS and BFV schemes bar bootstrapping. The library harnesses the power of NVIDIA GPUs through the Compute Unified Device Architecture (CUDA) [32] programming model. Our primary goal is to asses their performance on a recent GPU device with thousands of cores and excessive parallel computing capacity. We explain the details of the three methods, compare their timing and memory efficiencies and discuss advantages and/or disadvantages of each method. Our results indicate that, although Method III is generally recommended for timing efficiency, our results demonstrate that the method we refer to as Method II is more advantageous on the GPU, particularly for large ring sizes and large coefficient moduli. Furthermore, we highlight the inefficiencies of Method III in leveled variants of HE schemes, as well as the additional key requirements for each level. The key switching options offered in HEonGPU Library deliver up to a ×380 speedup compared to the same operation performed on a high-performance CPU using the Microsoft SEAL implementation.

The rest of the paper is organized as follows. In Section 2, we provide background information on the mathematical foundations, including polynomial rings, their arithmetic, and the residue number systems. We also explain the basic operations of the CKKS and BFV schemes. Additionally, in Section 2, we describe the implementation of the external product algorithm used for the relinearization process, focusing on Method I. Section 3 offers a more detailed explanation of the relinearization process using Method II and Method III. We present the specifics of our GPU implementation and, in Section 4, provide and discuss the implementation results. Finally, Section 5 concludes the paper by summarizing the findings of our work.

2 Preliminaries

This section presents the notation used throughout the paper and explains multiplication in the polynomial $\mathcal{R}_Q$ using number theoretic transform (NTT), as to how residue number system (RNS) enables a

[1] `https://github.com/Alisah-Ozcan/HEonGPU`

fast implementation and helps noise control. We also give the background on the functions of BFV and CKKS HE schemes and introduce the basic method for relinearization.

2.1 Notation

Let $N, Q \in \mathbb{Z}^+$ and N is a power of 2. $\mathbf{R}_Q \in \mathbb{Z}_Q / \langle X^N + 1 \rangle$, meaning the set of polynomials with integer coefficients of degree less than N, where the coefficients are in modulo Q. We define a polynomial $a = \sum_{i=0}^{N-1} a_i \cdot X^i \in \mathbf{R}_Q$ as vector of coefficient $(a_0, a_1, a_2, \ldots, a_{N-1}) \in \mathbb{Z}_Q^N$. We use the notation, $|a|_Q$, to define the reduction of an integer in modulo Q. $\|\mathbf{a}\|$ represents ℓ_∞-norm of coefficient vector of a. $\lceil \cdot \rfloor$, $\lfloor \cdot \rfloor$, and $\lceil \cdot \rfloor$ indicates rounding up, rounding down, indicates, and rounding to the nearest integer, respectively. The symbols $+$,$-$ and $\times$ (or just $\cdot$) denote addition, subtraction, and multiplication, respectively, in either $\mathbb{Z}_Q$ or $\mathbf{R}_Q$. By $a \xleftarrow{\$} S$, we denote that a is uniformly sampled from the finite set S. Finally, $e \leftarrow \chi$ denote a random sampling from the distribution χ, which is usually specified with mean and standard deviation values, μ and σ, respectively.

2.2 Multiplication in $\mathbf{R}_Q$ with Number Theoretic Transform

To compute $c = a \cdot b$, where $a, b, c \in \mathcal{R}_q$ we can use NTT, which is a form of Discrete Fourier Transform (DFT), in $\mathcal{O}(N \log(N))$. Using the negacylic NTT algorithm, the polynomial multiplication can be performed as follows

$$c = INTT(NTT(a) \odot NTT(b) \bmod q), \tag{1}$$

where $\odot$ stands for the element-wise (or Hadamard) multiplication of $NTT(a)$ and $NTT(b)$, which are vectors of N elements, each. The N-point negacyclic NTT and negacyclic INTT operations can be defined as follows:

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j \psi^{i \times j} \mod q, \;\; i \in \{0, 1, 2, \ldots, m-1\} \tag{2}$$

$$a_i = \sum_{j=0}^{n-1} \bar{a}_j \psi^{-i \times j} \mod q, \;\; i \in \{0, 1, 2, \ldots, m-1\} \tag{3}$$

The definition of negacyclic NTT requires the existence of a NTT friendly prime $q \in \mathbb{Z}^+$ where $q \equiv 1 \mod 2N$ and a constant value ($2N$-th root of unity) $\psi \in \mathbb{Z}_q$. ψ has to satisfy both conditions $\psi^{2N} \equiv 1 \mod q$ and $\psi^i \neq 1 \mod Q \;\; \forall i < 2N$.

When a polynomial $a \in \mathbf{R}_q$ is represented using its coefficients, it is said to be in the coefficient domain. If it is represented by the vector $\bar{a} = NTT(a)$, then we say it is in the NTT or the value domain.

2.3 Residue Number System (RNS)

The Chinese Remainder Theorem (CRT) suggests that an integer $X \in [0, Q)$ can be written with Eq. 4,

$$|X|_Q = \left| \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i \right|_Q, \tag{4}$$

where $Q = \prod_{i=0}^{\ell-1} q_i$, $x_i = X \bmod q_i$, and $Q_i = \frac{Q}{q_i}$ for $i \in [0, \ell-1]$. Here, the set of pairwise coprime integers $\mathfrak{B}_Q^\ell = \{q_0, q_1, \ldots, q_{\ell-1}\}$, is known as the base in the RNS representation of large integers. Also, the reduction with respect to moduli in the RNS base maps integers to $[-q_i/2, q_i/2]$, an operation often referred as the centered reduction. Alternatively, $X \in [0, Q)$ can be represented using Eq. 5 for some integer κ.

$$X = \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i - \kappa \cdot Q, \tag{5}$$

where we have,

$$\sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i < \ell \cdot Q. \tag{6}$$

In particular, as the sum exceeds Q by at most $(\ell-1) \cdot Q$, we have $\kappa \leq \ell - 1$.

Any integer $X \in \mathbb{Z}_Q$ has a unique RNS representation via its residues by the moduli in the base vector $\mathfrak{B}_Q^\ell$. As a result, operations such as addition, subtraction, multiplication, and division modulo M

can be performed concurrently with smaller moduli in $\mathfrak{B}_Q^\ell$ without using multi-precision arithmetic if the elements of $\mathfrak{B}_Q^\ell$ are chosen as single-precision integers. Computing residues with respect to $\mathfrak{B}_Q^\ell$, known as RNS-based decomposition, is illustrated in Eq. 7.

$$h(X) \mapsto (x_0, x_1, \ldots, x_{\ell-1}), \text{ where } x_i = X \bmod q_i, \tag{7}$$

where $h : \mathcal{R}_Q \mapsto \mathcal{R}_\ell$ is known as decomposition function. The inverse operation, which retrieves X, is known as composition, for which we use Eq. 4, and is denoted as $X = h^{-1}(x_0, x_1, \ldots, x_{\ell-1})$. RNS-based operations can be summarized as follows:

- **Modular Addition/Subtraction:** Given two numbers $X, Y \in \mathcal{R}_Q$ represented in RNS by their residues $[x_0, x_1, \ldots, x_{\ell-1}]$ and $[y_0, y_1, \ldots, y_{\ell-1}]$ respectively, the calculation of $Z = X \pm Y \bmod Q$ is performed as $z_i = x_i \pm y_i \mod q_i$, where $i = 0, 1, \ldots, \ell - 1$.
- **Multiplication:** In the same manner, $Z = X \times Y \bmod Q$ is calculated as $z_i = x_i \times y_i \mod q_i$ where $i = 0, 1, \ldots, \ell - 1$.
- **Division:** If $\gcd(Y, Q) = 1$, $Z = X \div Y \bmod Q$ is calculated as $z_i = x_i \times (Y)_{q_i}^{-1} \mod q_i$ where $i = 0, 1, \ldots, \ell - 1$.

Also, RNS supports operations such as base extension and base conversion. In base extension, an integer $X < Q$ in $\mathfrak{B}_Q^\ell$ can be represented in a larger base $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}} = \mathfrak{B}_Q^\ell \cup \{q_\ell, q_{\ell+1}, \ldots, q_{\tilde{\ell}-1}\}$, where $\tilde{Q} = Q \times \prod_{i=\ell}^{\tilde{\ell}-1} q_i$. To this end, the residues of X with respect to each new modulus are calculated $x_i = X \bmod q_i$ for $i = \ell, \ldots, \tilde{\ell} - 1$.

Base conversion, on the other hand, transforms an RNS representation of an integer into another, by changing the base vector, namely $\mathfrak{B}_Q^\ell \to \mathfrak{B}_{P'}^{r'}$, where $\mathfrak{B}_{P'}^{r'} = \{p'_0, \ldots, p'_{r'-1}\}$ and $P' = \prod_{i=0}^{r'-1} p'_i$. For accuracy of all integers in $\mathbb{Z}_Q$, we should have $Q \leq P'$. However, depending on the range of integers represented one can also consider conversion to a smaller base.

Both base extension and base conversion operations necessitate the accurate calculation of the parameter κ in Eq. 5. There are two methods to compute this parameter: i) the exact calculation and ii) the approximate calculation. The exact calculation method, first proposed in [33] [34], can be described as follows. The residue x_{re} with respect to the modulus q_{re} is computed as

$$x_{re} = |X|_{q_{re}} = \left| \left| \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i \right|_{q_{re}} - |\kappa \cdot Q|_{q_{re}} \right|_{q_{re}}, \tag{8}$$

where q_{re} is known as the redundant modulus and $\kappa \leq q_{re}$. If x_{re} is known (or pre-calculated) in addition to the residues in the main RNS base, we can determine the value of κ, which is essential to perform both base extension and conversion. To this end, Eq. 8 can be first rearranged into

$$|\kappa \cdot Q|_{q_{re}} = \left| \left| \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i \right|_{q_{re}} - |X|_{q_{re}} \right|_{q_{re}}. \tag{9}$$

When both sides of Eq. 9 are multiplied by $\left|Q^{-1}\right|_{m_{re}}$, we obtain

$$|\kappa|_{q_{re}} = \left| \left|Q^{-1}\right|_{q_{re}} \cdot \left(\left| \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i \right|_{q_{re}} - |X|_{q_{re}} \right) \right|_{q_{re}} \tag{10}$$

As $\kappa \leq q_{re}$, we finally have

$$\kappa = \left| \left|Q^{-1}\right|_{q_{re}} \cdot \left(\left| \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i \right|_{q_{re}} - |X|_{q_{re}} \right) \right|_{q_{re}} \tag{11}$$

Consequently, if an additional modulus q_{re} is kept alongside those in $\mathfrak{B}_Q^\ell$, κ can be exactly calculated, and thus the residues with respect to all moduli in both $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$ or $\mathfrak{B}_{P'}^{r'}$. The operation is repeated for every modulus in the new RNS base.

While this method enables the exact calculation of κ, its operational cost can be high due to the requirement of a redundant modulus. Alternatively, the approximate method can be employed, which

has a minimal error rate [35]. Similar to the previous method, the objective of this method is to calculate the value of κ for new moduli in the base extension and base conversion operations. Instead of using a redundant base, the approximate method utilizes floating-point arithmetic. The residue $|X|_{q_{new}}$ for a new modulus q_{new} can be written as

$$x_{new} = |X|_{q_{new}} = \left| \left| \sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i \right|_{q_{new}} - |\kappa \cdot Q|_{q_{new}} \right|_{q_{new}}. \tag{12}$$

Then, we have

$$\kappa = \left\lceil \frac{\sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i} \cdot Q_i}{Q} \right\rfloor. \tag{13}$$

since $Q_i = \frac{Q}{q_i}$, Eq. 13 can be rearranged as

$$\kappa = \left\lceil \frac{\sum_{i=0}^{\ell-1} \left| x_i \cdot Q_i^{-1} \right|_{q_i}}{q_i} \right\rfloor. \tag{14}$$

Once κ is determined, x_{new} can be calculated by using Eq. 12. Naturally, since floating-point arithmetic is employed, certain error is introduced in the calculations. However, the error rate can be minimized with appropriate base selection. Further details regarding the error rate are available in [35].

2.3.1 External Product and Gadget Decomposition

There are two important operations during homomorphic computations, which are explained in this section. The **External Product** is defined as the function $\mathcal{R}_Q \times \mathcal{R}_{\tilde{Q}}^{\ell} \rightarrow \mathcal{R}_Q$, where $\tilde{Q} = Q \cdot P$ and is represented with $\boxdot$ operator. Let $a \in \mathcal{R}_Q$ and $\mathbf{u} = [u_1, u_2, \ldots, u_\ell] \in \mathbf{R}_{\tilde{Q}}^{\ell}$, the external product of a and $\mathbf{u}$ defined as $\tilde{c} = a \boxdot \mathbf{u} = \sum_{i=1}^{\ell} b_i \cdot u_i \bmod \tilde{Q}$ and $c = \lceil \frac{1}{P} \cdot \tilde{c} \rfloor \bmod Q$, where $h(a) \mapsto \mathbf{b} = (b_1, b_2, \ldots, b_\ell)$. The **Gadget Decomposition**, used here, is essentially the RNS decomposition, which is mentioned in above as the function $h(x) : \mathcal{R}_Q \mapsto \mathcal{R}^{\ell}$, where for the gadget vector $\mathbf{g} = (g_0, g_1, \ldots, g_{\ell-1}) \in \mathcal{R}_Q^{\ell}$ we have $g_i = \left| Q_i^{-1} \right|_{q_i} \cdot Q_i$. Consequently, For $a \in \mathcal{R}_Q$ and its decomposition $h(a) \mapsto (a_0, a_1, \ldots, a_{\ell-1})$, we have

$$a = \sum_{i=0}^{\ell-1} a_i \cdot g_i \bmod Q \tag{15}$$

2.4 Homomorphic Encryption

Homomorphic encryption (HE) is a cryptographic system that enables operations on encrypted data without the need for decryption. BFV [6] and CKKS [9] are two of the most commonly used homomorphic encryption (HE) schemes. Despite their fundamentally different encryption, and decryption methods, both cryptosystems require a *'key switching'* operation for rotation and relinearization, the latter of which is necessary after homomorphic multiplication operations. Before delving into the details of the switch key operation, it is essential to provide a brief overview of the basic operations in both HE schemes. Let $Q = \prod_{i=0}^{\ell-1} q_i$, $P = \prod_{i=0}^{\pi-1} p_i$ and $\tilde{Q} = Q \cdot P$, where $\tilde{\ell} = \ell + \pi$. Note also that $q_i \leq P$ for $\forall q_i$ [6].

SecretKeyGeneration(λ):

$$sk = s, \text{ where } s \xleftarrow{\$} \mathcal{R}_2. \tag{16}$$

PublicKeyGeneration(sk):

$$\boldsymbol{pk} = (p_0, p_1) = (|-(a \cdot s + e)|_{\tilde{Q}}, a), \text{ where } (a \xleftarrow{\$} \mathcal{R}_{\tilde{Q}}), e \leftarrow \chi \tag{17}$$

EvaluationKeyGeneration(sk):

$$\boldsymbol{evk_i} = \left(\left| -a' \cdot s + s' \cdot P \cdot \frac{Q}{q_i} \cdot \left| \left(\frac{Q}{q_i}\right)^{-1} \right|_{q_i} + e' \right|_{\tilde{Q}}, a' \right) \tag{18}$$

for $0 \leq i < \ell - 1$, where $s = \boldsymbol{sk}$, $a' \xleftarrow{\$} \mathcal{R}_{\tilde{Q}}$, $e' \leftarrow \chi$. Here, $\boldsymbol{evk}$ can be relinearization, key switching or rotation keys. If it is a relinearization key, then $s' = s^2$ or a higher power of the secret key, depending on

the ciphertext size. For key switching, it is the new key while for rotation, a permutation of the secret key, $s' = \phi(s)$. Also, note that each part of the evaluation key is written in RNS representation with $\mathfrak{B}^{\tilde{\ell}}_{\tilde{Q}}$. Finally, the term $\frac{Q}{q_i} \cdot \left|(\frac{Q}{q_i})^{-1}\right|_{q_i}$ is the RNS gadget for $\mathcal{R}_Q$ and included for RNS composition with respect to Q,Since the explanation of the external product in this paper will be presented in the context of relinearization, the evaluation keys can be rewritten as

$$rlk_i = \left(\left|-a' \cdot s + s^2 \cdot P \cdot g_i + e'\right|_{\tilde{Q}}, a'\right) for i = 0, \dots \ell - 1. \tag{19}$$

Encryption($\boldsymbol{pk}, m$):

[label=–]**BFV:**

$$\boldsymbol{ct} = \left(\left|\Delta \cdot m + \frac{|p_0 \cdot u + e_1|_{\tilde{Q}}}{P}\right|_Q, \left|\frac{|p_1 \cdot u + e_2|_{\tilde{Q}}}{P}\right|_Q\right), \tag{20}$$

where $m \in \mathcal{R}_t$, $u \xleftarrow{\$} \mathcal{R}_2$ and $e_1, e_2 \leftarrow \chi$. **CKKS:**

$$\mathbf{ct} = \left(\left|\acute{m} + \frac{|p_0 \cdot u + e_1|_{\tilde{Q}}}{P}\right|_Q, \left|\frac{|p_1 \cdot u + e_2|_{\tilde{Q}}}{P}\right|_Q\right), \tag{21}$$

where $m \in \mathbb{R}$, $\acute{m} = \text{encode}(m)$, $u \xleftarrow{\$} \mathbf{R}_2$ and $e_1, e_2 \leftarrow \chi$.

Note that we only show the message encoding of the CKKS scheme, which is slightly different than that of BFV as it encodes messages that are real numbers.

Decryption($sk, \boldsymbol{ct}$):

[label=–]**BFV:**

$$\left\|\left\lceil \frac{t}{Q} \cdot |ct[0] + ct[1] \cdot s|_Q \right\rfloor\right\|_t \tag{22}$$

CKKS:

$$|ct[0] + ct[1] \cdot s|_Q \tag{23}$$

Note that we do not show the message decoding operations, which should be normally applied after the decryption operations.

Addition($\boldsymbol{ct}_0, \boldsymbol{ct}_1$): $(ct_0[0] + ct_1[0], ct_0[1] + ct_1[1])$.

Multiplication($\boldsymbol{ct}_0, \boldsymbol{ct}_1$): $\boldsymbol{ct}_2 = \boldsymbol{ct}_0 \times \boldsymbol{ct}_1$

[label=–]**BFV:**

$$\begin{aligned} ct_2[0] &= \left\|\left\lceil \frac{t}{Q} \cdot ct_0[0] \cdot ct_1[0] \right\rfloor\right\|_Q \\ ct_2[1] &= \left\|\left\lceil \frac{t}{Q} \cdot (ct_0[0] \cdot ct_1[1] + ct_0[1] \cdot ct_1[0]) \right\rfloor\right\|_Q \\ ct_2[2] &= \left\|\left\lceil \frac{t}{Q} \cdot ct_0[1] \cdot ct_1[1] \right\rfloor\right\|_Q \end{aligned} \tag{24}$$

CKKS:

$$\begin{aligned} ct_2[0] &= |ct_0[0] \cdot ct_1[0]|_Q \\ ct_2[1] &= |ct_0[0] \cdot ct_1[1] + ct_0[1] \cdot ct_1[0]|_Q \\ ct_2[2] &= |ct_0[1] \cdot ct_1[1]|_Q \end{aligned} \tag{25}$$

2.4.1 Relinearization Key Generation

Relinearization is an operation used to eliminate the nonlinear term, $ct[2]$, which arises after homomorphic multiplication as seen in Eq. 24 and Eq. 25. This requires the encrypted version of the term s^2, known as the relinearization key ($\boldsymbol{rlk}$), which is evaluation key when $s' = s^2$; i.e., $\boldsymbol{rlk} = \boldsymbol{evk}$. There are two established methods for performing relinearization [6]. The first is dynamic relinearization [6]; however, in this work, we will utilize the second method, referred to as 'modulus switching', which aims to minimize both the time and space required for relinearization. The **EvaluationKeyGeneration** and the **Encryption** functions described above are also formulated according to the second method. The fundamental principle of this method is that instead of the relinearization key working modulo Q, it works modulo $\tilde{Q}$. Consequently, s^2 is scaled accordingly as shown in the **EvaluationKeyGeneration** function.

The relinearization operation is performed as follows

$$ct_2[0] = \left\lfloor ct_2[0] + \left\lceil \frac{ct_2[2] \cdot \boldsymbol{rlk}[0]}{P} \right\rfloor \right\rfloor_Q, \tag{26}$$

$$ct_2[1] = \left\lfloor ct_2[1] + \left\lceil \frac{ct_2[2] \cdot \boldsymbol{rlk}[1]}{P} \right\rfloor \right\rfloor_Q, \tag{27}$$

which involves essentially three main operations: i) Number Theoretic Transform (NTT) (and its inverse iNTT), ii) Hadamard product, and iii) modular division for RNS representation. We can focus on the multiplication $ct_2[2] \cdot \boldsymbol{rlk}[i]$ where $ct_2[2] \in \mathcal{R}_Q$ and $\boldsymbol{rlk}[i] \in \mathcal{R}^{\ell}_{\tilde{Q}}$ for $i = 0, 1$. Direct multiplication is not possible as they are represented in two different RNS bases. We assume that $ct_2[2]$ is in the coefficient domain while $\boldsymbol{rlk}[i]$ in the NTT domain as the latter is pre-computed. Recalling that $ct_2[2]$ is represented in the RNS base with $\mathfrak{B}^{\ell}_{Q}$, we can decompose each element of its RNS representation with respect to $\mathfrak{B}^{\tilde{\ell}}_{\tilde{Q}}$, resulting in $\ell \times \tilde{\ell}$ polynomials[2]. Consequently, they can be multiplied by the relinearization keys $\boldsymbol{rlk}[i]$ and composed into the RNS base $\mathfrak{B}^{\tilde{\ell}}_{\tilde{Q}}$ thanks to the way the relinearization keys are prepared. Before the multiplication, however, we have to transform $ct_2[2]$ into the NTT domain, which requires $\ell \times \tilde{\ell}$ NTT operations. As the obtained result is in $\mathcal{R}_{\tilde{Q}}$, which is in the NTT domain, we need to convert it to polynomial domain by $2 \times \tilde{\ell}$ iNTT operations. Finally, all other operations such as division by P are performed in the coefficient domain. This method is the most classical method, which we refer here as the **Method I**.

3 Other Two Methods for Improved Relinearziation Operation

To reduce the number of NTT operations, which contributes to the execution time significantly and enhance the efficiency of the key-switching operation, two methods are proposed in the literature. In this section, we expound these two methods, which we refer here as **Method II** and **Method III**.

3.1 Method II

In Method II [29], the primes constituting Q can be combined into larger moduli by multiplying several of them. For instance, if $Q = \prod_{i=1}^{\ell} q_i$ with a base vector $\mathfrak{B}^{\ell}_{Q} = [q_1, q_2, \ldots, q_\ell]$, combining them allows it to be written as $Q = \prod_{i=1}^{d} D_i$ with a base vector $\mathfrak{B}^{d}_{Q} = [D_1, D_2, \ldots, D_d]$, where D_i is the multiplication of distinct subsets of the elements in $\mathfrak{B}^{\ell}_{Q}$. Note that we have the condition $D_i \leq P$ for $\forall D_i$ [28].

Recall that $h(x) : \mathcal{R}_Q \mapsto \mathcal{R}^{\ell}$ and $\tilde{h}(x) : \mathcal{R}_{\tilde{Q}} \mapsto \mathcal{R}^{\tilde{\ell}}$ are RNS gadgets decompositions corresponding to gadget vectors $\boldsymbol{g} = (g_1, g_2, \ldots, g_\ell) \in \mathbf{R}^{\ell}_{Q}$ and $\tilde{\boldsymbol{g}} = (\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_{\tilde{\ell}}) \in \mathcal{R}^{\tilde{\ell}}_{\tilde{Q}}$, respectively. The external product of $a \in \mathcal{R}_Q$ and $\boldsymbol{u} = (u_i)_{0 \leq i < \ell} \in \mathcal{R}_{\tilde{Q}}$ is computed as

$$a \boxdot \mathbf{u} = \sum_{i=0}^{\ell-1} b_i \cdot u_i \bmod \tilde{Q},$$

[2]In other words, we compute decomposition $\tilde{h}(b_i)$ for $i = 0, \ldots, \ell - 1$. As b_i values are already decomposed with $h(a)$, this decomposition operation is usually idempotent.

where $\mathbf{b} = h(a)$. Note that a and $\boldsymbol{u}$ are the nonlinear ciphertext $ct_2[2]$ and the relinearization key, $\boldsymbol{rlk}$, respectively, for the discussion here. As $\boldsymbol{rlk}$ consists of $rlk[0]$ and $rlk[1]$, the operation described in this section is repeated twice. As each u_i is represented in RNS form with respect to the modulus $\tilde{Q}$, we can write

$$a \boxdot \mathbf{u} = \sum_{i=0}^{\ell-1} b_i \cdot \sum_{j=0}^{\tilde{\ell}-1} v_{i,j} \cdot \tilde{g}_j \bmod \tilde{Q}.$$

If we can decompose each b_i with respect to $\tilde{g}$, then we obtain

$$a \boxdot \mathbf{u} = \sum_{i=0}^{\ell-1} \sum_{j=0}^{\tilde{\ell}-1} \beta_{i,j} \cdot v_{i,j} \cdot \tilde{g}_j \bmod \tilde{Q},$$

where $\beta_{i,j} = b_i \bmod q_j$ for $j = 0, \ldots \tilde{\ell} - 1$ or $\beta_{i,j} = \tilde{h}(b_i)$ for $i = 0, \ldots, \ell - 1$. When $b_i < q_j$ for $i = 0, \ldots, \ell - 1$ and $j = 0, \ldots, \tilde{\ell} - 1$, we can arrange the equation into

$$a \boxdot \boldsymbol{u} = \sum_{j=0}^{\tilde{\ell}-1} \left(\sum_{i=0}^{\ell-1} b_i \cdot v_{i,j} \right) \cdot \tilde{g}_j \bmod \tilde{Q}. \tag{28}$$

To perform $\ell \times \tilde{\ell}$ polynomial multiplications $b_i \cdot v_{i,j}$, we need to convert every b_i into the NTT domain with respect to each q_j for $j = 0, \ldots, \tilde{\ell} - 1$. This results in the computation of $\ell \times \tilde{\ell}$ NTT operations, which is the bottleneck in the computation of the external product.

When $\mathfrak{B}_Q^d$ is used, the decomposition function can be re-defined as $h_d(x) : \mathcal{R}_Q \mapsto \mathcal{R}^d$. Then, Eq. 28 becomes,

$$a \boxdot \mathbf{u} = \sum_{j=0}^{\tilde{\ell}-1} \left(\sum_{i=0}^{d-1} b_i \cdot v_{i,j} \right) \cdot \tilde{g}_j \bmod \tilde{Q}, \tag{29}$$

where $b = h_d(a)$. To perform the inner products $\langle b \cdot v_j \rangle = \sum_{i=0}^{d-1} b_i \cdot v_{i,j}$ using NTT, we need to base convert each $b_i \in \mathcal{R}_{D_i}$ for $i = 0, \ldots d - 1$ to the RNS base $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$. As a result, while it decreases the number of NTT operations to $\tilde{\ell} \times d$, Method II introduces d base conversions $\mathfrak{B}_{D_i} \mapsto \mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$ for $i = 0, \ldots d - 1$.

Note that since the result of $a \boxdot \boldsymbol{u}$ is in the NTT domain represented with the RNS base $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$, we need $2 \times \tilde{\ell}$ inverse NTT operations, which is the same as the one in Method I.

3.2 Method III

To decrease the number of NTT operations further, the authors in [30] propose a technique that groups primes both in $\mathfrak{B}_Q^{\ell}$ and $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$ into larger moduli such that $Q = \prod_{i=0}^{d-1} D_i$ and $\tilde{Q} = \prod_{j=0}^{\tilde{d}-1} \tilde{D}_j$, where $d < \ell$ and $\tilde{d} < \tilde{\ell}$. Here, $D_i = \prod_{k \in I_i} q_k$ and $\tilde{D}_j = \prod_{k \in \tilde{I}_j} \tilde{q}_k$ where $I_i \subset \{0, 1, \ldots \ell - 1\}$ and $\tilde{I}_j \subset \{0, 1, \ldots \tilde{\ell} - 1\}$, respectively. Also, we can re-define the gadget decomposition as $h_d(x) : \mathcal{R}_Q \mapsto \mathcal{R}^d$ and $h_{\tilde{d}}(x) : \mathcal{R}_{\tilde{Q}} \mapsto \mathcal{R}^{\tilde{d}}$. Consequently, we can rewrite Eq. 28 as

$$a \boxdot \boldsymbol{u} = \sum_{j=0}^{\tilde{d}-1} \left(\sum_{i=0}^{d-1} b_i \cdot v_{i,j} \right) \cdot \tilde{g}_j \bmod \tilde{Q}, \tag{30}$$

where a and $\boldsymbol{u}$ are decomposed into b_i and $v_{i,j}$ using the new gadget decomposition maps h_d and $h_{\tilde{d}}$, respectively. Then, one needs to perform $d \times \tilde{d}$ polynomial multiplications in $\mathcal{R}$.

However, if we want to use the NTT method for the polynomial multiplication, we need to adopt a different approach. In Method II, both b_i and $v_{i,j}$ are represented in the base $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$, for which all base elements are chosen NTT-friendly primes. As a result, in Method II the inner products $\langle \mathbf{b} \cdot \mathbf{v}_j \rangle$ are performed in modulo $\tilde{Q}$ for $j = 0 \ldots, \tilde{\ell}$. However, when we estimate the size of these inner products, we can easily observe that the result is much smaller than $\tilde{Q}$. Therefore, we can work with much smaller RNS base.

Let B and $\tilde{B}$ be the upper bounds for the coefficients of b_i and $v_{i,j}$, respectively. Then, we have $B = \frac{1}{2} \max_{1 \le j \le d} \{D_j\}$ and $\tilde{B} = \frac{1}{2} \max_{1 \le j \le \tilde{d}} \{\tilde{D}_j\}$ due to the centered reduction. Next, we obtain the upper bound B' for the infinity norm of the inner product as

$$\| \langle \mathbf{b} \cdot \mathbf{v}_j \rangle \|_\infty \le \| \mathbf{b} \|_\infty \cdot \| \mathbf{v}_j \|_\infty \le d \cdot N \cdot B \cdot \tilde{B} < P', \tag{31}$$

where $P' = \prod_{i=0}^{r'-1} p'_i$ for a certain set of primes p'_i. Subsequently, we can write $\langle \boldsymbol{b} \cdot \boldsymbol{v}_j \rangle \in \mathcal{R}_{P'}$. Naturally, we can write the result of the inner product in the RNS with the base vector $\mathfrak{B}_{P'}^{r'} = \{p'_0, p'_1, \ldots, p'_{r'-1}\}$. To this end, we need to convert each b_i from $\mathfrak{B}_{D_i}$ to $\mathfrak{B}_{P'}^{r'}$ dynamically while u_i can be pre-computed in the new RNS base. If we elect to work with ν-bit primes p'_i for the base $\mathfrak{B}_{P'}^{r'}$, then we calculate r' as

$$r' \geq \left\lceil \frac{\log(2dNB\tilde{B})}{\nu} \right\rceil \tag{32}$$

due to Eq. 31. If $r' < \tilde{\ell}$, the new RNS base decreases the number of NTT operations from $\ell \times \tilde{\ell}$ to $d \times r'$. Note that in Method III, we also group the primes in $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$, then we base convert $\boldsymbol{v}_j$. This conversion can be done in the precomputation stage.

Method III introduces more base conversion and iNTT operations. After the completion of the inner products $\langle \boldsymbol{b} \cdot \boldsymbol{v}_j \rangle$, which requires $2 \times d \times \tilde{d} \times r'$ Hadamard products, the results are in the RNS base $\mathfrak{B}_{P'}^{r'}$. Thus, we need to perform $2 \times \tilde{d} \times r'$ iNTT operations. Finally, we need to perform $2\tilde{d}$ base conversions $\mathfrak{B}_{P'}^{r'} \mapsto \mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$.

3.3 Comparison of Three Methods for External Product

Table 1 below compares the numbers of NTT, INTT, Hadamard product, and base conversion operations required for all three methods. The numbers of both NTT operations and Hadamard products decrease in Method II while the method incurs extra base conversion operations. As the based conversion is not necessarily an expensive operation, Method II is expected to accelerates the external product significantly for large value of N and Q. On the other hand, the overhead of Method III is more substantial. While it further decreases the number of NTT operations, it may significantly increase the numbers of Hadamard product, iNTT and base conversion operations depending on the values of $\tilde{\ell}$, $\tilde{d}$, and r'. In a CPU implementation of Method II reported in [30] for large values of N and Q, the authors report that Method III leads to significant speedup values over Method II. The advantage of Method III heavily depends on the extent the forward NTT operations dominates the execution time of the external product. As we show in the next section, where we present our GPU implementation results of all three methods, its advantage may disappear when NTT operations are accelerated using the parallel architecture of GPU devices.

Table 1: The number of NTT, iNTT, Hadamard product, and base conversion Operations

Method	NTT	INTT	H.P.	Base Conv
I	$\ell \times \tilde{\ell}$	$2 \times \tilde{\ell}$	$2 \times \ell \times \tilde{\ell}$	-
II	$d \times \tilde{\ell}$	$2 \times \tilde{\ell}$	$2 \times d \times \tilde{\ell}$	d
III	$d \times r'$	$2 \times \tilde{d} \times r'$	$2 \times d \times \tilde{d} \times r'$	$d + 2\tilde{d}$

4 Experimental Results and Comparison

Table 2: System Specifications

Feature	CPU	GPU
Model	Ryzen 9 7950X3D	GeForce RTX 4090
# of Cores	16(32 Threads)	16384
Frequency	4.20 GHz	2.520 GHz
RAM	128 GB	24 GB
Bandwidth	-	1.01 TB/s

- **Operating system and version:** Ubuntu 22.01
  CUDA version: 12.4

This section presents the results of the GPU implementations of three relinearization methods, based on approaches described in previous sections. These implementations, available within the HEonGPU Li-

brary[3], are compared both against each other and with the Microsoft SEAL implementation of Method I. Unlike prior work [30], where only CKKS versions of Methods II and III were evaluated, this study implements all three external product methods for both BFV and CKKS schemes, demonstrating execution time results across various levels in the leveled CKKS implementation. Method II shows substantial speedup for larger modulus Q and N, although its advantage diminishes as computations progress, often requiring a switch to Method I at higher levels. Meanwhile, Method III presents specific limitations, necessitating careful application. System specifications for these experiments, including both GPU and CPU configurations, are shown in Table 2 to illustrate the GPU's performance advantage over Microsoft SEAL, a highly optimized CPU implementation of BFV and CKKS schemes.

CUDA optimization strategies are thoroughly applied to the relinearization operations in HEonGPU to reduce memory latency and optimize computational resource utilization. Using fast memory hierarchies, such as registers and shared memory, minimizes global memory access time, while coalesced memory access patterns further enhance memory bandwidth efficiency. Instruction-Level Parallelism (ILP) enables the concurrent execution of independent instructions, maximizing processing unit efficiency during the relinearization computation. Carefully optimized thread and block configurations ensure optimal GPU occupancy and balanced workload distribution. The design philosophy emphasizes minimal kernel launches to reduce overhead, a critical strategy for large data size. Furthermore, HEonGPU Library leverages a state-of-the-art GPU Number Theoretic Transform (NTT) [36] available on GitHub,[4] which includes batch NTT support for polynomial artihmatic.

Given that BFV and CKKS follow distinct approaches, they utilize different parameters such as the size of primes and RNS bases. In BFV, essentially the ratio of $\frac{Q}{t}$ determines the noise budget. Hence, after multiplication and relinearization operations, BFV can perform additional multiplications if the budget allows, without requiring further operations. Hence, it is efficient to use as few primes as possible, where all primes should fit in word size of the computer. Therefore, the SEAL library always uses 59-bit primes for the RNS base $\mathfrak{B}_Q^\ell$. Naturally, we use large primes for our RNS in our BFV implementations.

Conversely, as CKKS operates on a leveled basis, after multiplication and relinearization operations, CKKS necessitates rescaling (modulus division) operation, thereby enabling further multiplications by dividing the error in the least significant bits. In CKKS, precision is affected by the bit lengths of the primes in the RNS base $\mathfrak{B}_Q^\ell$. Also, the number of primes in the RNS base determines the multiplicative depth of the circuit under homomorphic evaluation. This basically means that one can compromise precision for circuit depth or vice versa. Therefore, CKKS can use different number of primes in $\mathfrak{B}_Q^\ell$ for the same Q depending on the desired circuit depth or precision. In our implementation results, we sometimes report for two different precision of the CKKS scheme.

4.1 Execution Timing Results

Recall that I_i and $\tilde{I}_j$ are distinct subsets of primes in the RNS bases $\mathfrak{B}_Q^\ell$ and $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$, respectively. Namely, $D_i = \prod_{k\in I_i} q_k$ and $\tilde{D}_j = \prod_{k\in \tilde{I}_j} q_k$ for $i = 0, \ldots, d-1$ and $j = 0, \ldots, \tilde{d}-1$. For sake of simplicity, we assume $I = |I_i| = |\tilde{I}_j|$ for all values of i and j. In particular, $D_i = \prod_{k=0}^{I-1} q_{iI+k}$. Recall also that the term r' denotes the number of primes in the new base $\mathfrak{B}_{P'}^{r'}$ in Method III, namely $P' = \prod_{i=0}^{r'-1} p'_i$, where ν-bit primes are used in $\mathfrak{B}_{r'}$. As small values of r' decrease the number of NTT operations (see Table 1), we prefer working with larger single precision primes in $\mathfrak{B}_{r'}$. In particular, we employ $\nu = 60$ in our implementations.

Table 3 presents and compares the timings of three relinearization methods for the CKKS scheme for three different ring dimensions, $\log N \in \{14, 15, 16\}$ commonly used for homomorphic encryption applications, and up to five different values of $I \in \{2, 3, 4, 5, 6\}$. We do not present the timing results for lower ring dimension values such as $N \in \{2^{12}, 2^{13}\}$ as neither Method II nor Method III offers any advantage over Method I. Similarly, we do not include timings for all values of I when Q is the product of only few number of primes (i.e., small values of ℓ). In the table, the "Precision" column indicates the bit lengths of the moduli in the RNS bases $\mathfrak{B}_Q^\ell$ and $\tilde{\mathfrak{B}}_{\tilde{Q}}^{\tilde{\ell}}$. The "Level" column indicates the depth of the circuit that can be homomorphically evaluated as the employed CKKS scheme is leveled. Particularly, Level $\leq \ell - 1$, as one prime q_i should be dropped off from $\mathfrak{B}_Q^\ell$ after every level. That there are more than one levels for each ring dimension and that the number of levels increases with N in Table 3 calls for detailed explanation.

The security level of the lattice-based schemes is mainly determined by the values of N and Q. But, in the practical implementations of the CKKS and BFV schemes, we encrypt the public keys using a

[3] `https://github.com/Alisah-Ozcan/HEonGPU`

[4] `https://github.com/Alisah-Ozcan/GPU-NTT`

Table 3: Comparison of Execution Times of Three Relinearization Methods for CKKS on GPU (in μs)

Scheme	$\log N$	Precision	Level		Method I	Method II	Method III	Speedup
CKKS	14	32 bit	8	$(I, \tilde{\ell}, \ell, r')$	(-,10,9,-)	(2,11,9,-)	(2,11,9,5)	1.02 / 0.62
				Time	97.4	95.6	157.4	
			7	$(I, \tilde{\ell}, \ell, r')$	(-,9,8,-)	(3,11,8,-)	(3,11,8,7)	0.98 / 0.71
				Time	84.1	85.7	118.8	
	15	40 bit	18	$(I, \tilde{\ell}, \ell, r')$	(-,20,19,-)	(2,21,19,-)	(2,21,19,5)	1.72 / 1.49
				Time	903.7	525.1	605.8	
			17	$(I, \tilde{\ell}, \ell, r')$	(-,19,18,-)	(3,21,18,-)	(3,21,18,7)	1.95 /1.48
				Time	792.9	405.6	537.0	
			16	$(I, \tilde{\ell}, \ell, r')$	(-,18,17,-)	(4,21,17,-)	(4,21,17,9)	1.93 / 1.37
				Time	651.1	336.7	474.4	
			15	$(I, \tilde{\ell}, \ell, r')$	(-,17,16,-)	(5,21,16,-)	(5,21,16,11)	1.80 / 1.23
				Time	571.2	316.8	465.7	
			14	$(I, \tilde{\ell}, \ell, r')$	(-,16,15,-)	(6,21,15,-)	(6,21,15,13)	1.54 / 0.92
				Time	505.4	326.8	548.8	
		30 bit	23	$(I, \tilde{\ell}, \ell, r')$	(-,25,24,-)	(2,26,24,-)	(2,26,24,5)	1.69 / 1.75
				Time	1547.6	918.3	883.5	
			22	$(I, \tilde{\ell}, \ell, r')$	(-,24,23,-)	(3,26,23,-)	(3,26,23,7)	2.18 / 1.90
				Time	1420.2	652.1	746.0	
			21	$(I, \tilde{\ell}, \ell, r')$	(-,23,22,-)	(4,26,22,-)	(4,26,22,9)	2.32 / 1.86
				Time	1321.0	569.7	708.9	
			20	$(I, \tilde{\ell}, \ell, r')$	(-,22,21,-)	(5,26,21,-)	(5,26,21,11)	2.32 / 1.65
				Time	1213.4	522.3	733.5	
			19	$(I, \tilde{\ell}, \ell, r')$	(-,21,20,-)	(6,26,20,-)	(6,26,20,13)	2.24 / 1.52
				Time	1097.6	491.0	720.7	
	16	43 bit	37	$(I, \tilde{\ell}, \ell, r')$	(-,39,38,-)	(2,40,38,-)	(2,40,38,5)	1.75 / 1.64
				Time	7564.8	4315.9	4605.9	
			36	$(I, \tilde{\ell}, \ell, r')$	(-,38,37,-)	(3,40,37,-)	(3,40,37,7)	2.20 / 1.97
				Time	7129.7	3241.6	3611.2	
			35	$(I, \tilde{\ell}, \ell, r')$	(-,37,36,-)	(4,40,36,-)	(4,40,36,9)	2.60 / 2.32
				Time	6707.4	2581.8	2887.7	
			34	$(I, \tilde{\ell}, \ell, r')$	(-,36,35,-)	(5,40,35,-)	(5,40,35,11)	2.78 / 2.38
				Time	6337.0	2281.4	2666.5	
			33	$(I, \tilde{\ell}, \ell, r')$	(-,35,34,-)	(6,40,34,-)	(6,40,34,13)	2.72 / 2.20
				Time	5960.0	2194.8	2710.5	
		36 bit	43	$(I, \tilde{\ell}, \ell, r')$	(-,45,44,-)	(2,46,44,-)	(2,46,44,5)	1.80 / 1.77
				Time	10472.4	5837.7	5907.8	
			42	$(I, \tilde{\ell}, \ell, r')$	(-,44,43,-)	(3,46,43,-)	(3,46,43,7)	2.33 / 2.19
				Time	9946.5	4274.4	4548.0	
			41	$(I, \tilde{\ell}, \ell, r')$	(-,43,42,-)	(4,46,42,-)	(4,46,42,9)	2.73 / 2.49
				Time	9446.2	3458.4	3798.2	
			40	$(I, \tilde{\ell}, \ell, r')$	(-,42,41,-)	(5,46,41,-)	(5,46,41,11)	2.84 / 2.47
				Time	8956.3	3151.0	3619.9	
			39	$(I, \tilde{\ell}, \ell, r')$	(-,41,40,-)	(6,46,40,-)	(6,46,40,13)	3.03 / 2.59
				Time	8477.9	2800.9	3271.9	

larger modulus $\tilde{Q} = Q \times P$, where $P = \prod_{i=\ell}^{\tilde{\ell}-1} q_i$. Therefore, for the security level calculation we need to consider $\tilde{Q}$, which should not change for the targeted security level. Recall that $P > \forall q_i$ for Method I and additionally $P > \forall D_i$ in Methods II and III. Consequently, while a single prime P, which are larger than all values of q_i, is sufficient for Method I (i.e., $\tilde{\ell} = \ell + 1$), we may need to use multiple primes for Methods II and III depending on the value of I. When $\tilde{\ell} > \ell + 1$, we need to use a smaller Q, which also implies a reduced number of levels or a lower multiplicative depth.For instance, when $N = 2^{15}$ and $I = 4$ for 40-bit precision, the number of levels for Methods II and III will be reduced to 16 while for Method I it remains the highest possible value of 18 (see Table 3). Normally, for the same security level, all three methods should use the same $\tilde{Q}$. However, to ensure a fair comparison, we selected parameters that provide the same multiplication capacity, keeping Q constant while allowing $\tilde{Q}$ to vary based on the chosen P value. As a result, the security level provided by $\tilde{Q}$ in Method II and Method III is lower than the 128-bit security level achieved by Method I. Nevertheless, in all our experiments, we guarantee a minimum security level of approximately 110 bits for all configurations. It is worth reiterating that the parameters for Method II and Method III were selected with lower security levels solely to allow for a fair comparison with Method I.

Table 4: Comparison of Execution Times of Three Relinearization Methods for BFV on GPU (in μs)

Scheme	$\log N$	$\log Q$		Method I	Method II	Method III	Speedup
BFV	13	162 bit	$(I, \tilde{\ell}, \ell, r')$	(-,4,3,-)	(2,5,3,-)	(2,5,3,5)	0.84 / 0.61
			Time	25.6	30.3	42.0	
	14	381 bit	$(I, \tilde{\ell}, \ell, r')$	(-,8,7,-)	(2,9,7,-)	(2,9,7,5)	1.01 / 0.72
			Time	62.6	61.7	86.4	
	15	822 bit	(I, ℓ, ℓ, r')	(-,15,14,-) 492.5	(2,16,14,-)	(2,16,14,5)	1.63 / 1.21
			Time		302.2	405.8	
			$(I, \tilde{\ell}, \ell, r')$		(3,17,14,-)	(3,17,14,7)	1.86 / 1.25
			Time		265.0	394.5	
			$(I, \tilde{\ell}, \ell, r')$		(4,18,14,-)	(4,18,14,9)	1.92 / 1.19
			Time		256.4	412.5	
			$(I, \tilde{\ell}, \ell, r')$		(5,19,14,-)	(5,19,14,11)	1.98 / 1.20
			Time		248.7	409.1	
			$(I, \tilde{\ell}, \ell, r')$		(6,20,14,-)	(6,20,14,13)	1.74 / 0.98
			Time		282.7	503.6	
	16	1702 bit	(I, ℓ, ℓ, r')	(-,30,29,-) 4294.3	(2,31,29,-)	(2,31,29,5)	1.73 / 1.46
			Time		2478.1	2947.0	
			$(I, \tilde{\ell}, \ell, r')$		(3,32,29,-)	(3,32,29,7)	2.29 / 1.89
			Time		1877.7	2269.7	
			(I, ℓ, ℓ, r')		(4,33,29,-)	(4,33,29,9)	2.51 / 1.98
			Time		1708.1	2168.2	
			$(I, \tilde{\ell}, \ell, r')$		(5,34,29,-)	(5,34,29,11)	2.79 / 2.23
			Time		1541.9	1924.8	
			$(I, \tilde{\ell}, \ell, r')$		(6,35,29,-)	(6,35,29,13)	2.58 / 2.21
			Time		1667.1	1939.7	

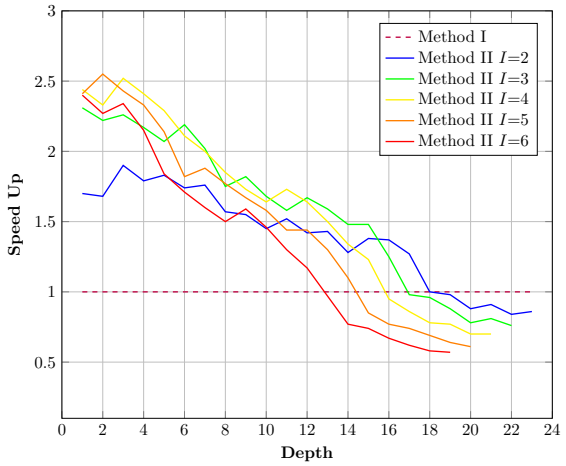
Table 5: Comparison of Relinearization timings for BFV and CKKS on CPU (Microsoft SEAL) and GPU (in μs)

Scheme	$\log N$	CPU	GPU			Speedup - (CPU Method I)/GPU		
		Method I	Method I	Method II	Method III	Method I	Method II	Method III
BFV	12	480	22.4	25.6	32.4	21.43	18.75	14.81
	13	1634	25.6	30.3	42.0	63.83	53.93	38.90
	14	12128	62.6	61.7	86.4	193.74	196.56	140.37
	15	76856	492.5	248.7	409.1	156.05	309.03	187.87
	16	588018	4294.3	1541.9	1924.8	136.93	381.36	305.50
CKKS	12	477	37.8	40.1	46.7	12.62	11.90	10.21
	13	1641	41.8	46.0	53.5	39.26	35.67	30.67
	14	14563	84.1	85.7	118.8	173.16	169.93	122.58
	15	169829	1321.0	569.7	708.9	128.56	298.10	239.57
	16	1215753	9446.2	3458.4	3798.2	128.70	351.54	320.09

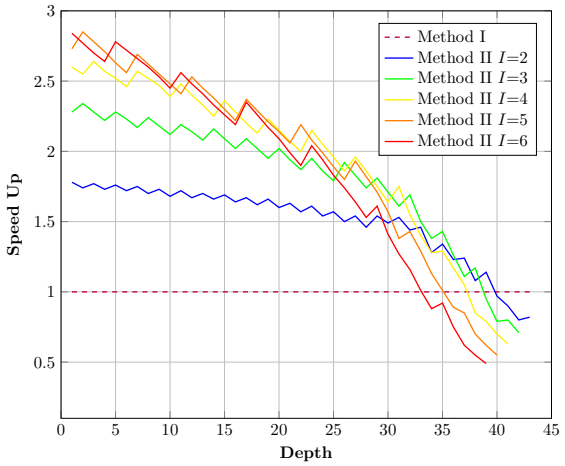
In the rightmost ("Speedup") column of Table 3, the speedup value of Methods II and III over Method I are given, respectively. Methods II and III offer speedup only for ring dimensions $N = 2^{15}$ and $N = 2^{16}$. For every configuration, Method II offers better performance than Method III except for one case when $N = 2^{15}$ for 30-bit precision and Level value of 23. One expected observation is that we generally obtain higher speedup values for larger values of I. However, the advantage disappear when $I \geq 5$ for relatively smaller values of ℓ (also larger values of r'). Another is that speedup values tend to be higher for the lower of the two precision values listed in the table. This is also expected as lower precision means higher ℓ values resulting in more NTT operations in Method I. This also means more room for Methods II and III to accelerate the relinearization operation.

In Table 4, we present the execution times of all three methods for the BFV scheme for four different ring dimensions $N \in \{2^{13}, 2^{14}, 2^{15}, 2^{16}\}$ and up to five different values of $I \in \{2, 3, 4, 5, 6\}$. As our BFV implementation is not leveled, we used a fixed size modulus Q for a ring dimension in our experiments to maintain the same noise budget. For this, we need to use larger $\tilde{Q}$ values for Methods II and III. For instance, in Table 4, when $N = 15$ and we use a 822-bit Q with $\ell = 14$ prime values of about 59 bits. While the value of $\tilde{\ell} = 15$ in Method I, $\tilde{\ell} \in [16, 20]$ in Methods II and III depending on the value of I. As in the case of CKKS, Method I provides higher level of security than the other two methods. In all our experiments, we ensure a minimum security level of about 110 bit.

In Table 4, we observe that Methods II and III provide speedup over Method I for ring dimensions of $N = 2^{15}$ and $N = 2^{16}$. We also observe that Method II offers more speedup values than Method III. The table suggests that we obtain better speedup values for $N = 2^{16}$ than $N = 2^{15}$, which is due to the fact that the former uses a higher value of ℓ. One expected observation is that higher values of I generally result in higher speedup values; however, up to a certain point. As increasing I also leads to a larger new RNS base (i.e., higher values of r'), one needs to find an optimum point, which may differ depending on the specific parameter set including N, Precision, the HE scheme.



(a) $N = 2^{15}$, 30-bit precision, and various I values. As the depth increases, performance declines due to the decreasing number of ℓ values.



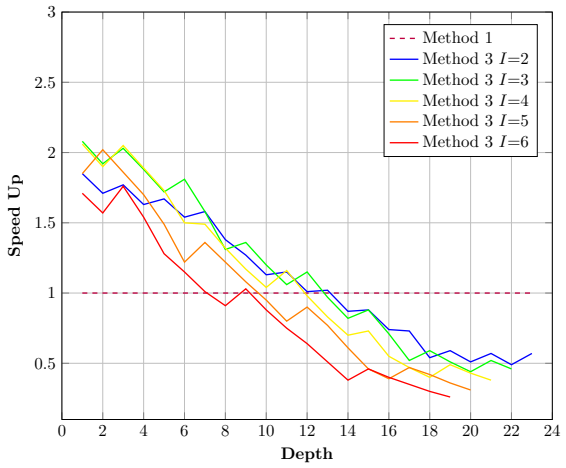
(b) $N = 2^{16}$, 36-bit precision, and various I values. As the depth increases, performance declines due to the decreasing number of ℓ values.

Figure 1: Change in the speed up values of Method II compared to Method I across all depths in CKKS.

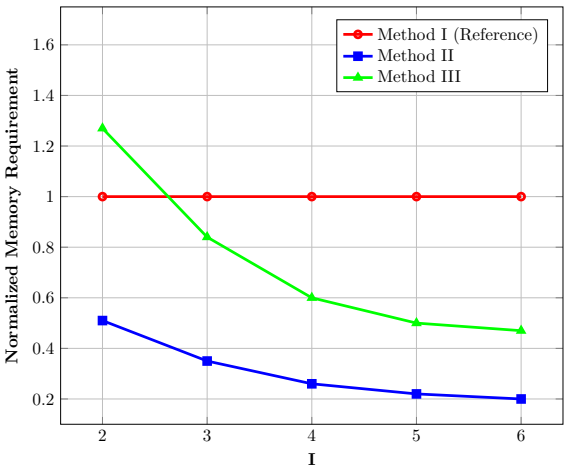
In Table 5, we provide the best GPU execution times of all three methods and their comparison to those of Microsoft SEAL Implementation of Method I with an identical parameter set running on our CPU. Table 5 suggests that GPU implementation can provide more than two orders of magnitude faster relinearization operation than a single-thread CPU implementation independent of the relinearization method used. We observe much higher speedup values for the larger values of N, which also means larger values of Q and $\tilde{Q}$. This is due to the fact that higher values of Q and $\tilde{Q}$ result in larger ℓ and $\tilde{\ell}$. This leads to higher number of NTT computations that can be performed in parallel on GPU. In Method I, the speedup values fall after $N = 2^{14}$ due to the excessive increase in the number of NTT operations, some of which are executed serially due to limitations on the GPU device. In Methods II and III, on the other hand, we always observe increase in speedup values. This can again be attributed to the reduction in the number of NTT operations. The speedup values confirm that both methods prove to be beneficial in parallel architectures where NTT operations can be performed concurrently.

In Figures 1a and 1b, we illustrate change in the speedup values of Method II over Method I obtained for the leveled CKKS scheme with $N = 2^{15}$ and $N = 2^{16}$ and 30- and 36-bit precision values for different levels homomorphic computation, respectively. The horizontal axis shows that the circuit has a multiplicative depth of around 20 and 40, respectively, for $N = 2^{15}$ and $N = 2^{16}$. Recall that after every homomorphic multiplication in the leveled CKKS scheme, one modulus from the RNS base is dropped due to the modulus switching operation. As can be observed from the figures, Method II becomes slower than Method I after certain number of homomorphic multiplications due to a very low values of ℓ. Therefore, one needs to consider switching from Method II to Method I during the computation of a deep circuit. Moreover, in Figure 2a , similar to the behavior observed in Figure 1a and 1b, we present the variation in speedup values of Method III compared to Method I for a leveled CKKS scheme with $N = 2^{15}$ and 30-bit precision, across different levels of homomorphic computation. As with Method II, it can be observed from the figures that due to very low ℓ values, Method III becomes slower than Method I after a certain number of homomorphic multiplications. However, Method III begins to exhibit inefficiencies at earlier levels compared to Method II. Therefore, it can be concluded that Method II is more efficient than Method III during the computation of a deep circuit.

In Figures 2b, the normalized key size with respect to Method I of three key switching methods (Method I, Method II, and Method III) are compared across different I values, with $N = 2^{16}$, for first level. Method I serves as the reference, maintaining a constant normalized key size regardless of I.



(a) This figure illustrates the performance of the Method 3 compared to the Method 1 across all depths in CKKS, for $N = 2^{15}$, 30 bit precision, and various I values. As the depth increases, performance declines due to the decreasing number of ℓ values.



(b) Normalized Key size Comparison of Three Different Key Switching Methods with respect to different I values, for $N = 2^{16}$.

Figure 2: Change in the speed up values of Method II compared to Method I across all depths in CKKS. Key size of Three Different Key Switching Methods.

This indicates a fixed memory cost that does not adapt to changes in I. On the other hand, Method II demonstrates significant efficiency gains as I increases, with its key size decreasing from 0.8 at I=2 to 0.2 at I=6. This behavior highlights its suitability for deep circuits, where low memory consumption is critical. Similarly, Method III exhibits a declining key size as I grows, starting at 1.2 for I=2 and decreasing to 0.6 for I=6. However, compared to Method II, Method III has a higher initial memory cost and a slower reduction rate, making it less efficient overall. Consequently, while Method I offers a consistent baseline and Method III provides moderate improvements, Method II emerges as the most memory-efficient approach, particularly for deeper circuits with larger I values. This makes Method II the preferred choice when memory consumption is a critical factor in homomorphic computation.

Finally, we depict the execution time ratios of the various operations in a relinearization operation for three methods for CKKS in Figure 3. Different colors in the bars indicate the execution rate of each operation such as NTT, iNTT, base conversion, Hadamard product (H.P.), modular division and the rest to the overall execution time of the relinearization operation. We group the bars of three methods depending on the value of I, where the leftmost group of bars is for $I = 2$ and its value increments from left to right until $I = 6$. The leftmost, the middle and the rightmost bars in each group correspond to Method I, Method II, and Method III, respectively. Even though always $I = 1$ for Method I, we include a different bar in each group for Method I, as its parameters has to match those of the other two methods in each group.

In Method I, NTT is always the dominant operation and a reduction in their number by means of Methods II and III predictably leads to significant acceleration in the computation of the relinearization operation. But, one curious observation is that the Hadamard product takes considerable portion of the execution times despite its low complexity nature. Two important factors account for this unexpected outcome: i) with its superior parallel computation power, GPU can execute NTT operation in much shorter amount of time and ii) therefore, the computation becomes more and more memory-bound as accessing coefficients of the polynomials in the global GPU memory consume a large portion of the execution time. This explains why further decrease in the number of NTT operation by Method III does not help accelerate the relinearization operation as the time spent on NTT becomes much less important. This is somewhat a different result than that in [30], where Method III outperforms Method II on a single threaded CPU where all NTT operations have to be performed sequentially. Therefore, whether Method II or Method III is a better option is subject to external factors such as parallel compute capability of the underlying target device. We expect that target platforms such as GPU and FPGA with extensive concurrent computation capacity, relinearization operation becomes more of memory or IO-bound than compute-bound. Therefore, we can claim that Method II tends to benefit much more when there is parallel computation capability.

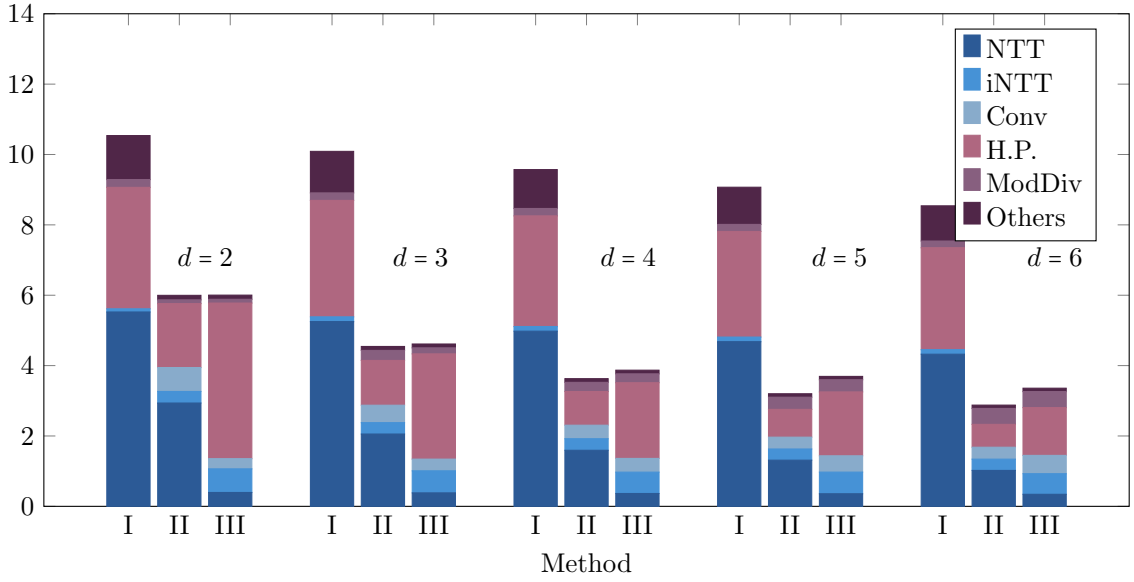


Figure 3: The execution ratios of each operation in the relinearizarion relative to d for $N = 2^{16}$ and 36-bit Precision. In the legend, 'H.P.', 'Conv', and 'ModDiv' denotes the Hadamard product, base conversion, and modulus division operations, respectively. Base conversions are for both $\mathbf{R}_{D_i} \rightarrow \mathbf{R}_{B'}$ and $\mathbf{R}_{B'} \rightarrow \mathbf{R}_{\tilde{D}_i}$.

Another disadvantage of Method III is that it requires a different relinearization key for each level in a leveled implementation of the HE scheme. As we also group the moduli in the RNS base $\mathfrak{B}_{\tilde{Q}}^{\tilde{\ell}}$, it will change from a level to the next as homomorphic computation progresses. Naturally, the grouping will also change which necessitates a different relinearization key. For instance, we cannot provide a leveled implementation of CKKS for $N = 2^{16}$ due to its memory requirements far exceeds the capacity of our GPU device.

4.2 Benchmarking Results for Hybrid Homomorphic Encryption

Hybrid Homomorphic Encryption (HHE) combines symmetric key encryption with HE. Data is encrypted using a symmetric key encryption algorithm to produce compact ciphertexts. The symmetric key is then encrypted using an FHE scheme, enabling homomorphic decryption of the ciphertext resulting in another ciphertext of the same message encrypted under the HE scheme. This hybrid design significantly reduces the ciphertext size compared to directly using FHE for data encryption, addressing one of the primary challenges of fully homomorphic systems.

Table 6: PASTA-3 Execution times in ms (single thread CPU and single GPU with default stream) and noise budget of the small HHE use case in the SEAL and HEonGPU libraries with different key-switching methods (security level = 128 bit).

Cipher	$\log N$	Plain Modulus	SEAL		HEonGPU				Speedup	
			Method I		Method I		Method II		SEAL vs HEonGPU	Method I vs Method II
			Time	R.N.B.	Time	R.N.B.	Time	R.N.B.	S_1	S_2
PASTA-3	14	65537	4677.28	96 bit	48.54	77 bit	44.79	37 bit	104.42	1.08
	15	65537	20404.89	525 bit	126.72	492 bit	106.28	445 bit	191.99	1.19
	16	786433	114632.33	1331 bit	713.20	1306 bit	493.73	1263 bit	232.17	1.44

R.N.B.: Remaining Noise Budget

In this work, we use homomorphic decryption of PASTA-3 symmetric key scheme [37] as a benchmark to compare the performances of Method I and Method II for a relatively complex homomorphic computation on GPU. The results are given in Table 6. The runtime results demonstrate a significant performance advantage of the HEonGPU library over the CPU implementation. For $\log N = 14$, the GPU implementation of Method II provides a speedup value of ×104.42 over CPU implementation. Also for $\log N = 14$, Method II is 1.08 faster than Method I on GPU. For a larger ring dimension, $\log N = 16$, the

advantage of the GPU implementation becomes more prominent, Method II on GPU is 232.17 times faster than Method I on CPU. Method II is now 1.44 faster than Method I on GPU for this ring dimension.

The remaining noise budget (R.N.B.), which indicates the noise budget left for further homomorphic operations after homomorphic decryption, depends on particular key-switching method, specific set of RNS base, and the library. CPU implementation of SEAL maintains a relatively higher R.N.B. due to its parameter choices. For instance, when $\log N = 14$, SEAL achieves 96 bits of R.N.B by utilizing $Q = 397$ bits and $P = 41$ bits with Method I. In the same case, HEonGPU achieves 77 bits of R.N.B by utilizing $Q = 378$ bits and $P = 60$ bits with Method I. The primary reason SEAL achieves a higher R.N.B is its use of a larger Q. As demonstrated in Table 6, in Method II, increasing P inherently reduces Q, which subsequently leads to a decrease in the R.N.B.

5 Conclusion

We evaluated the state-of-the-art GPU implementations of three existing key switching methods for the RNS variants of the CKKS and BFV schemes. Although time efficiency was our primary focus, we also asses their memory efficiency and their general perfomance in the leveled implementation of the CKKS scheme. Our detailed analysis indicate that Method II emerges as a stronger candidate for GPU implementations, particularly in scenarios involving large ring sizes and modulus coefficients. This is primarily due to its ability to better utilize the parallel processing power of GPUs. Moreover, our implementation demonstrates significant performance improvements, providing up to a 380× speedup in key switching operations compared to the CPU-based Microsoft SEAL library. Some of our results also suggest that there are cases or instances of key switching operations where Method I can be preferred such as the final set of homomorphic operations in the leveled computation. These results highlight the potential of GPUs, as well as other accelerators such as FPGAs and ASICs, for optimizing fully homomorphic encryption schemes. The findings underscore the growing importance of hardware-accelerated solutions in the development of high-performance homomorphic encryption libraries.

Acknowledgement

This work is supported by the European Union's Horizon Europe research and innovation programme under grant agreement No: 101079319.

References

[1] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology (EUROCRYPT)*. Springer, 1999, pp. 223–238.

[2] I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," in *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, ser. Lecture Notes in Computer Science, K. Kim, Ed., vol. 1992. Springer, 2001, pp. 119–136. [Online]. Available: https://doi.org/10.1007/3-540-44586-2_9

[3] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology (CRYPTO)*. Springer, 1985, pp. 10–18.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," Ph.D. dissertation, Stanford University, 2009.

[5] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," Cryptology ePrint Archive, Paper 2012/230, 2012, https://eprint.iacr.org/2012/230. [Online]. Available: https://eprint.iacr.org/2012/230

[6] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012, accessed pages 4, 6. [Online]. Available: https://eprint.iacr.org/2012/144

[7] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," Cryptology ePrint Archive, Paper 2012/078, 2012. [Online]. Available: https://eprint.iacr.org/2012/078

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014, accessed pages 4, 5.

[9] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437, accessed page 4.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," Cryptology ePrint Archive, Paper 2018/421, 2018, https://eprint.iacr.org/2018/421. [Online]. Available: https://eprint.iacr.org/2018/421

[11] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," Cryptology ePrint Archive, Paper 2014/816, 2014, https://eprint.iacr.org/2014/816. [Online]. Available: https://eprint.iacr.org/2014/816

[12] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," Cryptology ePrint Archive, Paper 2013/340, 2013, https://eprint.iacr.org/2013/340. [Online]. Available: https://eprint.iacr.org/2013/340

[13] J.-C. Bajard, J. Eynard, A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," Cryptology ePrint Archive, Paper 2016/510, 2016, https://eprint.iacr.org/2016/510. [Online]. Available: https://eprint.iacr.org/2016/510

[14] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," Cryptology ePrint Archive, Paper 2018/931, 2018, https://eprint.iacr.org/2018/931. [Online]. Available: https://eprint.iacr.org/2018/931

[15] "Concrete: Tfhe compiler that converts python programs into fhe equivalent," https://github.com/zama-ai/concrete, 2022, accessed page 3.

[16] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Heaan," https://github.com/snucrypto/HEAAN, 2016, accessed page 3.

[17] S. Halevi and V. Shoup, "Helib - an implementation of homomorphic encryption," https://github.com/shaih/HElib/, accessed Feb 2014.

[18] EPFL-LDS and T. I. SA, "Lattigo v5," Online: https://github.com/tuneinsight/lattigo, November 2023, accessed page 3.

[19] "Microsoft seal," https://github.com/Microsoft/SEAL, 2020, accessed page 3.

[20] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915. [Online]. Available: https://eprint.iacr.org/2022/915

[21] Y. Polyakov, R. Rohloff, G. W. Ryan, and D. Cousins, "Palisade lattice cryptography library (release 1.11.5)," https://palisade-crypto.org/, pp. 3, 4, 15, September 2021, https://gitlab.com/palisade/palisade-release/-/blob/master/doc/palisade_manual.pdf.

[22] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using gpu," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 2800–2803.

[23] A. Özcan, C. Ayduman, E. R. Türkoğlu, and E. Savaş, "Homomorphic encryption on gpu," *IEEE Access*, vol. 11, pp. 84 168–84 186, 2023.

[24] E. R. Türkoğlu, A. Özcan, C. Ayduman, A. C. Mert, E. Öztürk, and E. Savaş, "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.

[25] 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," Cryptology ePrint Archive, Paper 2021/508, 2021, https://eprint.iacr.org/2021/508. [Online]. Available: https://eprint.iacr.org/2021/508

[26] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2020.

[27] Y. Doröz, E. Öztürk, E. Savas, and B. Sunar, "Accelerating ltv based homomorphic encryption in reconfigurable hardware," in *CHES*. Springer, 2015, pp. 185–204. [Online]. Available: https://www.iacr.org/archive/ches2015/92930182/92930182.pdf

[28] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," Cryptology ePrint Archive, Paper 2020/1203, 2020, https://eprint.iacr.org/2020/1203. [Online]. Available: https://eprint.iacr.org/2020/1203

[29] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," Cryptology ePrint Archive, Paper 2021/204, 2021. [Online]. Available: https://eprint.iacr.org/2021/204

[30] M. Kim, D. Lee, J. Seo, and Y. Song, "Accelerating HE operations from key decomposition technique," Cryptology ePrint Archive, Paper 2023/413, 2023, https://eprint.iacr.org/2023/413. [Online]. Available: https://eprint.iacr.org/2023/413

[31] A. Şah Özcan and E. Savaş, "HEonGPU: a GPU-based fully homomorphic encryption library 1.0," Cryptology ePrint Archive, Paper 2024/1543, 2024. [Online]. Available: https://eprint.iacr.org/2024/1543

[32] NVIDIA Corporation, "Cuda toolkit documentation," https://developer.nvidia.com/cuda-toolkit, 2023, accessed May 2023.

[33] P. Trebicki and S. Grabowski, "Modular multiplication and base extensions in residue number systems," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, pp. 217–220.

[34] ——, "Fast base extension using a redundant modulus in rns," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1011–1014, 1993.

[35] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," Cryptology ePrint Archive, Paper 2018/117, 2018, https://eprint.iacr.org/2018/117. [Online]. Available: https://eprint.iacr.org/2018/117

[36] A. Şah Özcan and E. Savaş, "Two algorithms for fast gpu implementation of ntt," Cryptology ePrint Archive, Paper 2023/1410, 2023, https://eprint.iacr.org/2023/1410. [Online]. Available: https://eprint.iacr.org/2023/1410

[37] C. Dobraunig, L. Grassi, L. Helminger, C. Rechberger, M. Schofnegger, and R. Walch, "Pasta: A case for hybrid homomorphic encryption," Cryptology ePrint Archive, Paper 2021/731, 2021. [Online]. Available: https://eprint.iacr.org/2021/731