

Article

GPU Acceleration for KLSS Key Switching in Fully Homomorphic Encryption

Shutong Jin  and Ray C. C. Cheung *

Department of Electrical Engineering, City University of Hong Kong, Tat Chee Avenue, Hong Kong SAR, China; shutong.jin@my.cityu.edu.hk

* Correspondence: r.cheung@cityu.edu.hk

Abstract

Fully Homomorphic Encryption (FHE) enables privacy-preserving computation but is hindered by high computational overhead, with the key-switching operation being a primary performance bottleneck. This paper introduces the first CUDA-optimized GPU implementation of the Kim, Lee, Seo, and Son (KLSS) key-switching algorithm for three leading FHE schemes: BGV, BFV, and CKKS. Our solution achieves significant performance gains, delivering speedups of up to $181\times$ against the original CPU implementation. Furthermore, we analyze the critical trade-off between the key-switching techniques on GPUs, providing insights for the choice between single- and double-decomposition methods. Our work provides a high-performance tool and offers clear guidelines on the trade-off between latency and hardware memory constraints.

Keywords: fully homomorphic encryption; key switching; GPU acceleration**MSC:** 11T71

Academic Editors: Yanbin Li and Yongjun Ren

Received: 6 November 2025

Revised: 20 November 2025

Accepted: 25 November 2025

Published: 27 November 2025

Citation: Jin, S.; Cheung, R.C.C. GPU Acceleration for KLSS Key Switching in Fully Homomorphic Encryption. *Mathematics* **2025**, *13*, 3809. <https://doi.org/10.3390/math13233809>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Fully Homomorphic Encryption (FHE) represents a paradigm shift in cryptography by enabling computation directly on encrypted data without prior decryption. This capability allows for the processing of sensitive information by untrusted third parties, such as cloud service providers, while guaranteeing data confidentiality. Consequently, FHE is an immensely attractive technology for a wide range of privacy-preserving applications, including secure cloud computing, private machine learning inference, and federated learning [1–5].

Despite this promise, the practical deployment of FHE has been hindered by significant performance challenges. The substantial computational overhead associated with current FHE schemes remains a critical barrier to their widespread adoption in real-world systems. Among the various homomorphic operations, key switching is a particularly demanding primitive. It is a fundamental building block for other essential operations, such as ciphertext relinearization after multiplication and automorphisms. Given its frequent invocation and resource-intensive nature, the efficiency of the key-switching procedure has a profound impact on the overall performance of an FHE system.

This bottleneck has drawn considerable attention from the research community, leading to continuous efforts on developing more efficient key-switching algorithms. Among these, the technique by Kim, Lee, Seo, and Son (KLSS) [6] is particularly noteworthy. It reduces the complexity, measured in the number of Number Theoretic Transform (NTT) operations, from $\mathcal{O}(\ell^2)$ to $\mathcal{O}(\ell)$ with respect to the ciphertext level ℓ .

While the theoretical advantages of the KLSS algorithm are clear, its practical benefits depend heavily on an optimized implementation that can fully exploit modern hardware architectures. To date, however, implementations have often been scheme-specific or tailored to CPU environments.

In this work, we bridge this gap by presenting the first unified, GPU-accelerated implementation of the KLSS key-switching algorithm for three of the most prominent FHE schemes: BGV [7], BFV [8–10], and CKKS [11]. Our contributions are threefold:

1. A CUDA-optimized GPU implementation of KLSS key switching for the BGV, BFV, and CKKS schemes.
2. A comprehensive benchmark against CPU and prior state-of-the-art GPU implementations, demonstrating significant speedups.
3. An in-depth analysis of the trade-off between single- and double-decomposition for key switching, specifically within the context of GPU-accelerated homomorphic encryption.

The remainder of this paper is organized as follows. Section 2 reviews the necessary background on FHE and GPU acceleration. Section 3 looks into the specifics of the FHE schemes and the key-switching procedure relevant to our work. Section 4 details the architecture of our unified GPU implementation. Section 5 presents our experimental setup and performance results. Section 6 provides a detailed discussion of our findings. Finally, Section 7 concludes the paper and discusses potential directions for future work.

2. Related Works

2.1. Fully Homomorphic Encryption

Since Gentry’s blueprint in 2009 [12], the field of FHE has largely been built on lattice-based cryptography. The security of these systems is rooted in the computational hardness of the Learning with Errors (LWE) and Ring-LWE (RLWE) problems [7,9,10,12–16]. The schemes based on the RLWE problem, such as BGV [7], BFV [8–10], and CKKS [11], have gained prominence due to their high computational efficiency. This efficiency stems from their ability to pack multiple data values into a single ciphertext, enabling parallel processing via Single Instruction, Multiple Data (SIMD) operations. The BGV and BFV schemes are designed for exact computations over finite fields, while the CKKS scheme is tailored for approximate arithmetic on real and complex values. This makes CKKS particularly well-suited for applications in privacy-preserving machine learning [2,11,17–21]. The practical implementation and academic study of these schemes are supported by many open-source libraries, including SEAL [22], HELib [23], and OpenFHE [24].

2.2. Research in Key Switching

Previous research on key switching [25] produced two primary techniques with distinct trade-offs. The Gentry, Halevi, and Smart (GHS) method is highly efficient, requiring only a linear number of NTTs [16]. However, this efficiency comes at the cost of either halving the modulus size or doubling the polynomial dimension to preserve security. In contrast, the Brakerski and Vaikuntanathan (BV) technique is more direct to implement but incurs a higher computational cost, involving a quadratic number of NTTs [10].

Hybrid key switching was developed to combine the digit decomposition technique from BV with the larger modulus strategy from GHS to strike a better balance [26]. More recent research [27] has continued to build upon the hybrid approach to further minimize the computational overhead. For instance, some work focuses on optimizing the selection of decomposition parameters for different ciphertext levels [28]. A more recent advancement, now considered the state-of-the-art, is the KLSS technique, which significantly reduces the computational workload by further optimizing the use of NTTs [6].

2.3. GPU-Accelerated FHE

To mitigate the substantial computational demands of FHE, researchers have turned to Graphics Processing Units (GPUs) for acceleration. The inherent parallelism of FHE, especially the SIMD batching, makes GPUs a natural fit for this task, and numerous studies have confirmed their effectiveness in accelerating homomorphic operations [29–35].

The research landscape for GPU acceleration can be divided based on the type of FHE scheme. For bit-wise schemes, the open-source library cuFHE was one of the first to provide a CUDA-based implementation for TFHE [36]. Subsequent work on FHEW and TFHE has introduced advanced parallelization strategies and memory-aware optimizations tailored to their unique computational patterns [37–39].

Simultaneously, a significant body of research has focused on accelerating word-type schemes like CKKS, BFV, and BGV, which are critical for arithmetic-heavy applications. The central aim is to reduce the computational overhead that impedes practical FHE deployment. To achieve this, researchers have developed optimizations at multiple levels, from algorithmic enhancements in key switching to software strategies like kernel fusion that reduce memory latency [40–42]. Other efforts have targeted hardware-specific features, such as leveraging Tensor Cores for further acceleration [43,44]. Most recently, the Cheddar library was introduced, demonstrating state-of-the-art performance by integrating these multi-level optimization techniques [45].

Most open-source library, like HEonGPU [42], provide implementations for previous variants of key-switching techniques (BV [10], GHS [16], Hybrid [26]). To the best of our knowledge, no existing GPU acceleration library offers an implementation for KLSS key switching or other double-decomposition based methods, a gap that directly motivated this work.

3. Preliminaries

The security of the Learning With Errors (LWE) problem stems from the computational hardness of recovering a secret vector from a system of linear equations that contains a small amount of noise. Formally, for a given modulus q and dimension n , an LWE instance is a pair $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$. Here, $\mathbf{a}$ is a vector chosen uniformly at random, and b is derived as the inner product of $\mathbf{a}$ and a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, perturbed by a small error term $\mathbf{e}$:

$$b = \langle \mathbf{a}, \mathbf{s} \rangle + \mathbf{e} \pmod{q}.$$

The error $\mathbf{e}$ is typically drawn from a discrete Gaussian distribution.

The Ring Learning With Errors (RLWE) problem is the ring-based analogue of LWE, offering improved efficiency by operating over polynomial rings. Within a ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(f(x))$, the challenge is to find a secret polynomial $\mathbf{s} \in \mathcal{R}_q$. This is done using samples $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^2$, where $\mathbf{a}$ is a uniformly random polynomial and $\mathbf{b}$ is computed as

$$\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \pmod{q},$$

with $\mathbf{e}$ being an error polynomial with small coefficients.

In this work, we operate within the polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where N is a power of two. A ciphertext, denoted as $\mathbf{ct}$, is a pair of polynomials $(\mathbf{c}_0, \mathbf{c}_1)$. The correctness of decryption relies on the relationship

$$\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} = \mathbf{m} + \mathbf{e},$$

where $\mathbf{s}$ is the secret key, $\mathbf{m}$ is the encoding of the plaintext μ , and $\mathbf{e}$ is a small error term.

3.1. BGV, BFV, CKKS

We now summarize the core functionalities of three RLWE-based FHE schemes: BGV [7], BFV [8–10], and CKKS [11].

- **Setup:** The process begins by establishing the cryptographic environment based on a security parameter λ . This involves selecting public parameters $pp = (N, t, Q, P, \chi_{\text{key}}, \chi_{\text{err}})$, which define the polynomial ring dimension N , plaintext modulus t , ciphertext modulus Q , and the distributions for key and error sampling (χ_{key} and χ_{err}).
- **Key Generation:** A secret key $\mathbf{sk}$ is created by sampling a polynomial $\mathbf{s}$ from the key distribution χ_{key} . The corresponding public key $\mathbf{pk}$ is a pair $(\mathbf{a}, -\mathbf{a}\mathbf{s} + \mathbf{e})$, where $\mathbf{a}$ is a uniformly random polynomial in R_Q and $\mathbf{e}$ is a small error from χ_{err} . Additionally, evaluation keys, such as relinearization keys $\mathbf{ksk}_{\text{relin}}$ and automorphism keys $\mathbf{ksk}_{\text{auto}}$, are generated for homomorphic operations.
- **Encryption:** To encrypt a plaintext $\mathbf{m}$, it is first encoded into a polynomial $\mathbf{m}^*$. The specific encoding method depends on the scheme:

- **BFV:** The message is scaled to the most significant bits: $\mathbf{m}^* = \lfloor \frac{Q}{t} \rfloor \cdot [\mathbf{m}]_t$.
- **BGV:** A correction factor μ is applied: $\mathbf{m}^* = [\mu \mathbf{m}]_t$.
- **CKKS:** The message is directly encoded as a polynomial: $\mathbf{m}^* = \mathbf{m}$.

The final ciphertext $\mathbf{ct} = (\mathbf{c}_0, \mathbf{c}_1)$ is then formed by adding this encoded message to a fresh encryption of zero, which is calculated as $\text{Enc}_{pk}(0) = [\mathbf{r} \cdot (\mathbf{a}, \mathbf{b}) + \mathbf{t} \cdot (\mathbf{e}_0, \mathbf{e}_1)]_Q$.

- **Decryption:** The original message is recovered from a ciphertext using the secret key $\mathbf{s}$.
 - **BFV:** $\mathbf{m} = \lfloor \frac{t}{Q} \cdot [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_Q \rfloor$
 - **BGV:** $\mathbf{m} = [\mu^{-1} [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_Q]$.
 - **CKKS:** $\mathbf{m} = [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_Q$
- **Homomorphic Operations:** These schemes support computations directly on encrypted data.
 - **Addition:** The sum of two ciphertexts, $\mathbf{ct}$ and $\mathbf{ct}'$, is simply their component-wise sum: $\mathbf{ct}_{\text{add}} = \mathbf{ct} + \mathbf{ct}'$.
 - **Multiplication:** The product of two ciphertexts is computed, followed by a relinearization step using $\mathbf{ksk}_{\text{relin}}$ to produce $\mathbf{ct}_{\text{mult}}$.
 - **Automorphism:** An automorphism (e.g., rotation) is applied to a ciphertext $\mathbf{ct}$ using a corresponding key $\mathbf{ksk}_{\text{auto}}$ to yield the transformed ciphertext $\mathbf{ct}_{\text{auto}}$.

3.2. Key Switching

Key switching is a generic technique for changing the secret key under which a ciphertext is encrypted, from an initial key $\mathbf{s}_A$ to a target key $\mathbf{s}_B$, without altering the underlying plaintext. A switching key, $\mathbf{ksk}_{s_A \rightarrow s_B}$, is defined as:

$$\mathbf{ksk}_{s_A \rightarrow s_B} = ([\mathbf{s}_A + \mathbf{a} \cdot \mathbf{s}_B + \mathbf{te}]_Q, -\mathbf{a}) \in \mathcal{R}_Q^2$$

where $\mathbf{a}$ is a uniformly random polynomial in $\mathcal{R}_Q^2$ and $\mathbf{e}$ is a small error sampled from χ_{err} .

3.2.1. Relinearization

Relinearization is a specific application of key switching used after a homomorphic multiplication. The multiplication of two linear ciphertexts results in a quadratic ciphertext of degree 2, effectively encrypted under $(\mathbf{1}, \mathbf{s}, \mathbf{s}^2)$. Relinearization uses a key-switching key $\mathbf{ksk}_{\text{relin}}$ to convert this quadratic ciphertext back into a linear one under the original basis $(\mathbf{1}, \mathbf{s})$. Our implementation unifies the key-switching module for both relinearization and other operations (like automorphisms). For non-relinearization inputs, we structure the

input as $[c_0^*, 0, c_1^*]$, where the term to be switched, c_1^* , is treated as the third component (c_2) to maintain a consistent interface.

3.2.2. Modulus Switching

Modulus Switching is a technique for managing noise growth. It reduces the modulus of a ciphertext from a larger value Q to a smaller one Q' while preserving the plaintext. This is done by scaling the ciphertext coefficients by the ratio Q'/Q and rounding to the nearest integer:

$$ct' = \left\lfloor \frac{Q'}{Q} \cdot ct \right\rfloor \pmod{Q'}$$

4. GPU Acceleration for KLSS

4.1. The Double-Decomposition Approach

The KLSS key-switching technique [6] builds upon the prior state-of-the-art methods such as hybrid key switching, by introducing a double-decomposition strategy. As illustrated in Figure 1, after being raised to $Q_\ell P$, both the input ciphertext ct_A and the key-switching key ksk are additionally converted to an external base T for external product. In Section 4.2, we restructure the original Algorithm 1 to align with the massively parallel architecture of GPUs, thereby maximizing its computational throughput.

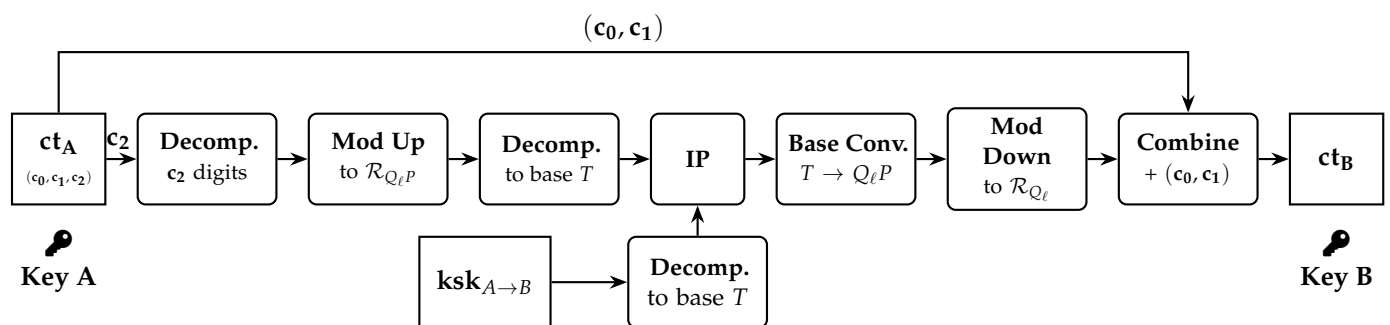


Figure 1. The computational workflow for key switching using the KLSS technique. Decomp: Digit Decomposition; IP: Inner Product.

4.2. GPU Acceleration

To better leverage the massively parallel architecture of modern GPUs, we restructured the original algorithm into a two-stage process. The first stage is an offline preprocessing step, where all necessary data is precomputed and loaded into GPU global memory. The second stage performs the online calculation of the key-switching ciphertext based on a given input. These two stages are detailed in Algorithms 2 and 3, respectively.

4.2.1. Stage 1: Key Generation (Offline)

This module is executed once. All intermediate and final key data are generated and stored in the GPU's global memory (Device memory) to avoid costly data transfers during the online phase.

4.2.2. Stage 2: GPU-Accelerated Key Switching (Online)

This module leverages the pre-computed key ksk_{double} from the GPU memory. The input c_i is first transferred to the GPU, all computations are performed on the device, and only the final result is transferred back.

Algorithm 1 KLSS Key Switching**Input:** Context parameters: Bases q, P, E ; Decompositions d, d_{double} .**Input: Offline:** Secret keys $\mathbf{s}_A, \mathbf{s}_B$.**Input: Online:** Ciphertext component $\mathbf{c}_i \in \mathcal{R}_q$.**Output:** Switched ciphertext parts $(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1) \in \mathcal{R}_{Q_\ell}$.**STAGE 1. Generates $\mathbf{ksk}_{double}$ and store.**

```

1: for  $l \leftarrow 0$  to  $d$  do
2:   Sample  $\mathbf{a}_l, \mathbf{e}_l$ . Compute raw parts:
3:    $\mathbf{ksk}_{0,l} \leftarrow [\mathbf{a}_l \mathbf{s}_B + t \mathbf{e}_l + P \cdot \mathcal{B}(\mathbf{s}_A)_l]_{Q_\ell P}$ 
4:    $\mathbf{ksk}_{1,l} \leftarrow [-\mathbf{a}_l]_{Q_\ell P}$ 
   // BaseConvert (Map from  $Q_\ell P \rightarrow T$ )
5: for  $j \in \{0, 1\}, l \in [0, d], m \in [0, d_{double} - 1]$  do
6:    $\mathbf{poly} \leftarrow \tilde{\mathcal{D}}(\mathbf{ksk}_{j,l})_m$  ▷ Extract digit
7:    $\mathbf{ksk}_{double,j,l,m} \leftarrow \text{NTT}_{\text{fwd}}(\text{BaseExt}(\mathbf{poly}, Q_{\tilde{m}}, T))$ 
   ▷ Store in NTT form in Ring  $\mathcal{R}_T$  for fast online dot products.

```

STAGE 2. Transforms $\mathbf{c}_i$ using $\mathbf{ksk}_{double}$.

1. Input Preparation (Decompose & BaseConvert)

```

8: for  $l \leftarrow 0$  to  $d$  do
9:    $\mathbf{poly} \leftarrow \mathcal{D}(\mathbf{c}_i)_l$  ▷ Decompose input  $\mathbf{c}_i$ 
10:   $\mathbf{c}_{\text{ext},l} \leftarrow \text{NTT}_{\text{fwd}}(\text{BaseExt}(\mathbf{poly}, q_T, T))$ 
   ▷ Move input parts to Ring  $\mathcal{R}_T$  and transform to NTT.

```

2. Inner Product

```

11: for  $j \in \{0, 1\}$  do
12:   Initialize accumulator  $\mathbf{acc}_j \leftarrow 0 \in \mathcal{R}_T$ .
13:   for  $m \leftarrow 0$  to  $d_{double} - 1$  do
14:      $\mathbf{temp} \leftarrow 0$ 
15:     for  $l \leftarrow 0$  to  $d$  do
16:        $\mathbf{temp} \leftarrow \mathbf{temp} + \mathbf{c}_{\text{ext},l} \cdot \mathbf{ksk}_{double,j,l,m}$  ▷ Pointwise Mult-Add
17:        $\mathbf{acc}_j \leftarrow \mathbf{acc}_j + \mathbf{temp} \cdot \text{NTT}_{\text{fwd}}(\tilde{\mathcal{B}}_m)$  ▷ Reconstruct in  $\mathcal{R}_T$ 
18:    $\mathbf{c}_j'' \leftarrow \text{NTT}_{\text{inv}}(\mathbf{acc}_j)$  ▷ Inverse NTT in Ring  $\mathcal{R}_T$ 

```

3. Post-Processing (BaseConvert & ModDown)

```

19: for  $j \in \{0, 1\}$  do
20:    $\mathbf{c}_j' \leftarrow \text{BaseExt}(\mathbf{c}_j'', T, Q_\ell P)$  ▷ Convert back to large modulus
21:    $\mathbf{ffi}_j \leftarrow -t \cdot \text{BaseExt}([\text{NTT}_{\text{inv}}(t^{-1} \mathbf{c}_j')]_{P, P, Q_\ell})$  ▷ Correction factor
22:    $\tilde{\mathbf{c}}_j \leftarrow [P^{-1}(\text{NTT}_{\text{inv}}(\mathbf{c}_j') + \mathbf{ffi}_j)]_{Q_\ell}$  ▷ Scale down to  $Q_\ell$ 
23: return  $(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$ 

```

Algorithm 2 GPU-Accelerated Key Generation for KLSS**Input:** New secret key $\mathbf{s}_B \in \mathcal{R}_{Q_\ell P}$, old secret key $\mathbf{s}_A' \in \mathcal{R}_{Q_\ell P}$ (on Device).**Output:** Double-decomposition key $\mathbf{ksk}_{double}$ stored in GPU Device memory.

// Step 1: Generate standard key parts in a batched kernel.

```

1: parallel for  $l \in [0, d - 1]$  do ▷ Launch a kernel to process all  $\omega$  parts in parallel.
2:   Sample a batch of  $d$  polynomials  $\mathbf{a}_l$  and  $\mathbf{e}_l$  on the GPU.
   ▷ All polynomial arithmetic is coefficient-wise and parallelized within the kernel.
3:  $\mathbf{ksk}_{0,l} \leftarrow [\mathbf{a}_l \cdot \mathbf{s}_B + t \mathbf{e}_l + P \cdot \mathcal{B}(\mathbf{s}_A)_l]_{Q_\ell P}$ 
4:  $\mathbf{ksk}_{1,l} \leftarrow [-\mathbf{a}_l]_{Q_\ell P}$ 

```

// Step 2: Apply second decomposition and base-extend in a single, large-scale kernel.

```

5: parallel for  $(j, l, m) \in \{0, 1\} \times [0, d - 1] \times [0, d_{double} - 1]$  do ▷ A massively parallel kernel.
   ▷ Each thread block be assigned to compute one or more  $\mathbf{ksk}_{double,j,l,m}$ .
6:   Decompose  $\mathbf{ksk}_{j,l}$  to get  $\tilde{\mathcal{D}}(\mathbf{ksk}_{j,l})_m$ . ▷ RNS decomposition is coefficient-wise.
7:   Let  $\mathbf{poly} = \tilde{\mathcal{D}}(\mathbf{ksk}_{j,l})_m$ .
8:    $\mathbf{ksk}_{double,j,l,m} \leftarrow \text{BaseExt}(\mathbf{poly}, Q_{\tilde{m}}, T)$ . ▷ Batched BaseExt with error correction.
9: return  $\mathbf{ksk}_{double}$  (residing on the GPU Device).

```

Algorithm 3 GPU-Accelerated Double-Decomposition KLSS Key Switching.**Input:** Ciphertext component $\mathbf{c}_i \in \mathcal{R}_{Q_\ell}$ (on Host), key $\mathbf{ksk}_{double}$ (on Device).**Output:** Switched ciphertext components $(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1) \in \mathcal{R}_{Q_\ell}^2$ (on Host).

```

1: Transfer  $\mathbf{c}_i$  from Host to GPU Device memory.
   // Step 1: Batched Input Extension.
2: Decompose  $\mathbf{c}_i$  into  $(\mathcal{D}(\mathbf{c}_i)_0, \dots, \mathcal{D}(\mathbf{c}_i)_{d-1})$  on the GPU.
3: parallel for  $l \in [0, d-1]$  do                                ▷ Launch a kernel for batched base extension.
4:  $\mathbf{c}_{ext,l} \leftarrow \text{BaseExt}(\mathcal{D}(\mathbf{c}_i)_l, q_l, T)$ .                                ▷ Batched BaseExt

   // Step 2: Batched Dot Product in  $\mathcal{R}_E$ .
5: parallel for  $(j, l, m) \in \{0, 1\} \times [0, d] \times [0, d_{double} - 1]$  do                                ▷ Batched NTTs and
   multiplications.
6:  $\mathbf{temp}_{j,l,m} \leftarrow \text{NTT}_{\text{fwd}}(\mathbf{c}_{ext,l}) \odot \text{NTT}_{\text{fwd}}(\mathbf{ksk}_{double,j,l,m})$ .    ▷  $\odot$  is coefficient-wise product.
7: parallel for  $(j, m) \in \{0, 1\} \times [0, d_{double} - 1]$  do                                ▷ Parallel reduction over index  $l$ .
8:  $\mathbf{c}_{j,m}' \leftarrow \sum_{l=0}^d \mathbf{temp}_{j,l,m}$ .                                ▷ Kernel using parallel reduction.
9: parallel for  $j \in \{0, 1\}$  do                                ▷ Parallel CRT reconstruction (another reduction).
10:  $\mathbf{c}_j'' \leftarrow \sum_{m=0}^{d_{double}-1} \mathbf{c}_{j,m}' \odot \text{NTT}_{\text{fwd}}(\tilde{\mathbf{B}}_m)$ .

   // Step 3: Batched Base Conversion from  $\mathcal{R}_T$  back to  $\mathcal{R}_{Q_\ell P}$ .
11: parallel for  $j \in \{0, 1\}$  do
12:  $\mathbf{c}_j' \leftarrow \text{BaseExt}(\text{NTT}_{\text{inv}}(\mathbf{c}_j''), T, Q_\ell P)$ .                                ▷ Batched  $\text{NTT}_{\text{inv}}$  and BaseExt.

   // Step 4: Batched Final Scaling.
13: parallel for  $j \in \{0, 1\}$  do
14:  $\mathbf{ffi}_j \leftarrow -t \cdot \text{BaseExt}([\text{NTT}_{\text{inv}}(t^{-1}\mathbf{c}_j')]_P, P, Q_\ell)$ .                                ▷ Batched kernels for all steps.
15:  $\tilde{\mathbf{c}}_j \leftarrow [P^{-1}(\text{NTT}_{\text{inv}}(\mathbf{c}_j') + \mathbf{ffi}_j)]_{Q_\ell}$ .
16: Transfer  $(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$  from GPU Device to Host memory.
17: return  $(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$ .

```

4.3. Unified Framework for BFV, BGV, and CKKS

While the original KLSS paper focused primarily on the CKKS scheme, we extended the implementation to encompass all word-type schemes, including BFV and BGV. By designing the framework illustrated in Figure 2, we developed a unified implementation that achieves comparable efficiency across all three schemes, as detailed in Tables 1 and 2. Additionally, we adapted our implementation to align with the original Phantom library, where ciphertexts for BGV and CKKS are stored in the NTT domain.

Table 1. Per ciphertext key switching latency for Hybrid and KLSS methods for BFV, BGV, and CKKS schemes on an A100 GPU. All latency values are in milliseconds (ms).

Set #	BFV			BGV			CKKS		
	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup
1	0.94	0.85	1.10×	1.03	0.97	1.06×	1.00	0.95	1.06×
2	0.61	0.68	0.89×	0.67	0.74	0.91×	0.66	0.72	0.92×
3	0.49	0.71	0.69×	0.55	0.81	0.67×	0.53	0.73	0.73×
4	1.36	1.19	1.15×	1.49	1.36	1.10×	1.48	1.32	1.12×
5	0.86	0.79	1.09×	0.96	0.89	1.08×	0.92	0.88	1.04×
6	0.64	0.79	0.80×	0.75	0.85	0.89×	0.71	0.87	0.81×
7	1.86	1.51	1.23×	1.93	1.63	1.18×	1.94	1.67	1.17×
8	1.14	1.03	1.11×	1.20	1.13	1.07×	1.17	1.12	1.05×
9	0.80	0.92	0.88×	0.89	1.03	0.87×	0.91	1.02	0.89×
10	5.58	3.30	1.69×	5.66	3.73	1.52×	5.67	3.62	1.57×
11	3.46	2.96	1.17×	3.55	3.23	1.10×	3.56	3.17	1.12×
12	2.48	2.29	1.09×	2.70	2.58	1.05×	2.70	2.53	1.07×
13	1.91	2.02	0.95×	2.11	2.26	0.94×	2.04	2.30	0.89×

Table 1. Cont.

Set #	BFV			BGV			CKKS		
	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup
14	8.62	4.41	1.95×	8.62	4.77	1.81×	8.54	4.72	1.81×
15	5.23	4.13	1.27×	5.39	4.53	1.19×	5.32	4.44	1.20×
16	3.90	2.87	1.36×	4.07	3.23	1.26×	4.01	3.29	1.22×
17	2.89	2.63	1.10×	3.09	2.91	1.06×	3.06	2.92	1.05×
18	12.00	5.81	2.06×	12.12	6.29	1.93×	12.17	6.27	1.94×
19	7.41	5.27	1.41×	7.52	5.68	1.32×	7.55	5.68	1.33×
20	5.22	3.73	1.40×	5.51	4.15	1.33×	5.38	4.15	1.30×
21	4.15	3.14	1.32×	4.33	3.62	1.20×	4.26	3.62	1.17×

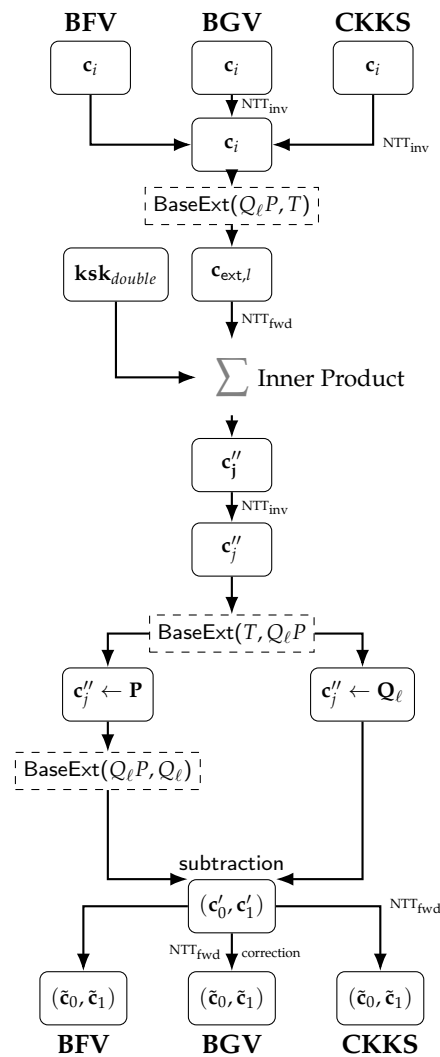


Figure 2. The unified framework to accommodate GPU-accelerated KLSS key switching for BFV, BGV, CKKS.

Table 2. Per-ciphertext key switching latency and performance comparison for Hybrid and KLSS methods across BFV, BGV, and CKKS schemes on an NVIDIA RTX 4090 GPU.

Set #	BFV			BGV			CKKS		
	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup
1	0.65	0.60	1.07×	0.71	0.67	1.06×	0.71	0.67	1.05×
2	0.37	0.45	0.83×	0.41	0.51	0.80×	0.40	0.50	0.80×
3	0.29	0.41	0.71×	0.34	0.47	0.72×	0.33	0.46	0.72×

Table 2. Cont.

Set #	BFV			BGV			CKKS		
	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup	Hybrid [26,41] (ms)	Ours (ms)	Speedup
4	0.91	0.85	1.07×	0.98	0.92	1.06×	0.97	0.91	1.06×
5	0.54	0.54	1.00×	0.58	0.60	0.97×	0.57	0.59	0.98×
6	0.39	0.47	0.83×	0.44	0.53	0.82×	0.42	0.53	0.79×
7	1.20	1.15	1.04×	1.29	1.24	1.04×	1.28	1.23	1.04×
8	0.73	0.70	1.04×	0.79	0.77	1.02×	0.78	0.76	1.03×
9	0.51	0.56	0.91×	0.57	0.64	0.90×	0.56	0.62	0.91×
10	3.84	3.03	1.27×	4.03	3.23	1.25×	4.04	3.21	1.26×
11	2.37	2.59	0.92×	2.52	2.75	0.92×	2.51	2.73	0.92×
12	1.74	1.88	0.93×	1.88	2.03	0.93×	1.86	2.02	0.92×
13	1.34	1.58	0.85×	1.46	1.76	0.83×	1.45	1.72	0.84×
14	6.18	4.15	1.49×	6.38	4.42	1.44×	6.38	4.40	1.45×
15	3.83	3.85	0.99×	3.90	4.07	0.96×	3.89	4.06	0.96×
16	2.86	2.56	1.12×	2.94	2.78	1.06×	2.91	2.77	1.05×
17	2.19	2.22	0.99×	2.26	2.43	0.93×	2.24	2.42	0.93×

5. Results

5.1. Experimental Setup

Our performance evaluation was conducted on two distinct hardware platforms: a consumer-grade NVIDIA RTX 4090 GPU and a datacenter-class NVIDIA A100 GPU. The A100 was specifically required for evaluating large parameter sets, as the substantial memory footprint of the associated key-switching keys exceeded the memory capacity of the consumer-grade RTX 4090. For our CPU performance baseline, we reference the results reported in [6]. Those experiments were performed on a system running Ubuntu 20.04.3 LTS, equipped with 192 GB of RAM and an Intel Xeon Platinum 8268 CPU (2.90 GHz), with each test utilizing a single thread. The specific parameters used for all benchmarks are detailed in Table 3.

Table 3. Summary of Parameters used for log N = 15 and log N = 16.

log N	Set #	$\tilde{l}$	ℓ	r	Optimal r_{double}
15	1	16	15	1	4
	2	16	14	2	5
	3	16	13	3	5
	4	20	19	1	4
	5	20	18	2	5
	6	20	17	3	5
	7	24	23	1	4
	8	24	22	2	5
	9	24	21	3	7
16	10	32	31	1	5
	11	32	30	2	3
	12	32	29	3	3
	13	32	28	4	6
	14	40	39	1	5
	15	40	38	2	3
	16	40	37	3	5
	17	40	36	4	6
	18	48	47	1	5
	19	48	46	2	3
	20	48	45	3	5
	21	48	44	4	6

5.2. GPU Acceleration Performance

As shown in Table 4, our GPU implementation achieves significant speedups, ranging from $84\times$ to $150\times$ on the NVIDIA A100 and from $114\times$ to $181\times$ on the NVIDIA RTX 4090 (including the host-device data transfer and kernel initialization). This performance difference is attributed to the distinct architectural trade-offs between A100 and RTX 4090. While the A100 features higher memory and bandwidth, the RTX 4090 has a substantially greater number of CUDA cores (16,384 versus 6912 for the A100). The higher speedup on the RTX 4090 indicates that our algorithm is predominantly compute-bound, and thus benefits more from the larger count of parallel processors than from the A100's superior memory bandwidth.

Table 4. Per ciphertext key switching latency of CPU vs. GPU Implementation (this work).

Set #	CPU [6] (ms)	NVIDIA A100		NVIDIA RTX 4090	
		Time (ms)	Speedup	Time (ms)	Speedup
1	84	0.95	$88.59\times$	0.67	$124.66\times$
2	62	0.72	$86.68\times$	0.50	$123.04\times$
3	66	0.73	$90.35\times$	0.46	$143.87\times$
4	112	1.32	$85.01\times$	0.91	$122.69\times$
5	85	0.88	$96.15\times$	0.59	$144.52\times$
6	87	0.87	$99.61\times$	0.53	$164.61\times$
7	140	1.67	$84.07\times$	1.23	$113.68\times$
8	110	1.12	$98.13\times$	0.76	$144.39\times$
9	101	1.02	$99.26\times$	0.62	$162.05\times$
10	451	3.62	$124.75\times$	3.21	$140.57\times$
11	352	3.17	$110.95\times$	2.73	$128.96\times$
12	335	2.53	$132.63\times$	2.02	$165.50\times$
13	301	2.30	$131.01\times$	1.72	$174.67\times$
14	604	4.72	$127.92\times$	4.40	$137.34\times$
15	508	4.44	$114.31\times$	4.06	$125.14\times$
16	479	3.29	$145.43\times$	2.77	$172.64\times$
17	438	2.92	$149.91\times$	2.42	$181.35\times$

5.3. Comparison with Hybrid Key Switching

A subject of ongoing discussion in the field is the relative performance of hybrid key-switching techniques (single decomposition) versus the KLSS algorithm [46] (double decomposition). We conduct a direct performance evaluation, comparing our implementation of the KLSS algorithm against a state-of-the-art, GPU-optimized hybrid method by Yang et al. [41]. To ensure a fair and direct comparison, we applied an identical GPU acceleration strategy to both algorithms. The results, presented in Tables 1 and 2, indicate that neither method demonstrates universal superiority.

Our analysis shows that the hybrid KLSS method, even when configured with optimal decomposition parameters, does not consistently outperform the simpler, more memory-efficient single-decomposition strategy. Specifically, the performance advantage of KLSS diminishes as the parameter r increases or as the multiplication depth ℓ decreases. This observation aligns with the theoretical insights provided by [46], although their experimental validation was confined to CPU architectures.

Conversely, the KLSS method demonstrates superior performance for parameter sets with a large multiplication depth ℓ and a small number of RNS decomposition word size r for the single-decomposition part. Under these favorable conditions, such as those of parameter Set #14, the KLSS approach achieves a speedup of up to $1.49\times$ over the single-decomposition strategy. A more detailed discussion of these trade-offs is presented in Section 6.

6. Discussion

6.1. Single- or Double-Decomposition? A Trade-Off Analysis

The choice between single and double decomposition represents a fundamental trade-off between computational latency and available multiplicative depth. Mono & Güneysu [46] recently contended that prior comparisons to hybrid key switching in [6] were inequitable. They demonstrated that for latency-optimized scenarios, a single-decomposition approach configured with a large gadget vector dimension r (and thus a decomposition base $d = 1$) can match or even surpass the performance of double decomposition. Our own experiments, detailed in Tables 1 and 2, corroborate this finding in a GPU-accelerated context: for shallow circuits where raw speed is paramount, single decomposition is indeed the superior choice.

However, this latency-centric perspective does not capture the full picture. We argue that the double decomposition method remains a crucial tool in scenarios where maximizing multiplicative depth is the primary constraint. This is particularly relevant for applications involving deep arithmetic circuits where bootstrapping is either unavailable or prohibitively expensive. In such cases, the number of available levels, ℓ , is a fixed resource that dictates the complexity of the computable functions.

The advantage of double decomposition lies in its management of the modulus chain. The total modulus is composed of primes for the ciphertext modulus Q and primes for the special modulus P used in key switching. The number of levels, ℓ , is determined by the number of primes composing Q . To achieve low latency with single decomposition, one must use a large r , which in turn requires a large P constructed from multiple prime moduli. These primes are thus consumed by P and are unavailable for Q , effectively reducing the maximum achievable depth ℓ .

In contrast, double decomposition allows for a much smaller special modulus P . By setting the inner gadget dimension $r = 1$, P can be constructed from a single prime from the modulus chain. This minimizes the resources allocated to key switching and maximizes the number of primes available for the ciphertext modulus Q . Consequently, for a given set of cryptographic parameters, double decomposition enables a greater multiplicative depth ℓ . Therefore, the recommendation by [46] to use $d = 1$ is not universally optimal. The choice of decomposition scheme is not absolute but rather a strategic decision dictated by application-specific requirements: single decomposition for speed in shallow circuits, and double decomposition for depth in complex, non-bootstrapped computations.

6.2. GPU Key Size Constraints

Beyond the latency-focused discussion in [46], a critical and often decisive factor for practical GPU implementations is the strictly limited nature of on-chip GPU memory (VRAM). To achieve the high throughput promised by GPU acceleration, the entirety of the evaluation keys must reside in VRAM to avoid performance-killing data transfers over the PCIe bus. The choice of decomposition scheme directly impacts the feasibility of this requirement.

This trade-off between decomposition parameters and memory consumption is vividly illustrated in the provided plot, which we will reference as Figure 3. The figure plots the key-switching key size against the required multiplication depth, ℓ , for various single- and double-decomposition configurations. Several clear trends emerge:

1. **Impact of Gadget Dimension:** The size of the gadget vector dimension, r (and r_{double} for double decomposition), has a dramatic effect on the total key size. In the single-decomposition case, increasing the dimension from $r = 2$ to $r = 8$ results in a significant increase in memory footprint.
2. **The Cost of Double Decomposition:** The penalty is even more pronounced with double decomposition. As shown in the figure, moving from 'Double-Decomp ($r = 3, r_{double}$)

$= 2)$ to $(r = 3, r_{\text{double}} = 8)$ causes a steep rise in key size. Crucially, for comparable parameters, double decomposition consistently requires more memory than single decomposition.

The key sizes shown, reaching up to 0.5 GB, are already substantial. However, it is vital to recognize that these values represent a best-case scenario. The plot likely assumes a moderate polynomial degree N . In practice, higher security requirements often necessitate larger parameters like $N = 65,536$, which would immediately quadruple the key sizes shown. Table 5 shows the maximum size for a single key-switching key with the different techniques. With double decomposition, a single key-switching key can demand up to 145 GB in the worst case. Furthermore, Figure 3 only accounts for the key switching (relinearization) key. A complete application requires a full set of Galois keys for operations like vector rotations, and the total size of these keys can easily be an order of magnitude larger than the relinearization key alone. Figure 4 further illustrates the dramatic growth of key size with r_{double} , while the corresponding efficiency gains fail to keep pace.

When these scaling factors are considered, a key size of 0.5 GB can easily inflate to 20–40 GB or more. This brings the memory requirement into direct conflict with the VRAM capacity of many common GPUs (e.g., 24 GB on an NVIDIA RTX 4090). If the total key size exceeds available VRAM, the system is forced to swap key data between system RAM and VRAM, creating an I/O bottleneck that negates the computational advantages of the GPU. Therefore, while double decomposition may offer greater multiplicative depth, its associated memory cost can render it impractical for GPU acceleration, making single decomposition the only viable option in memory-constrained environments.

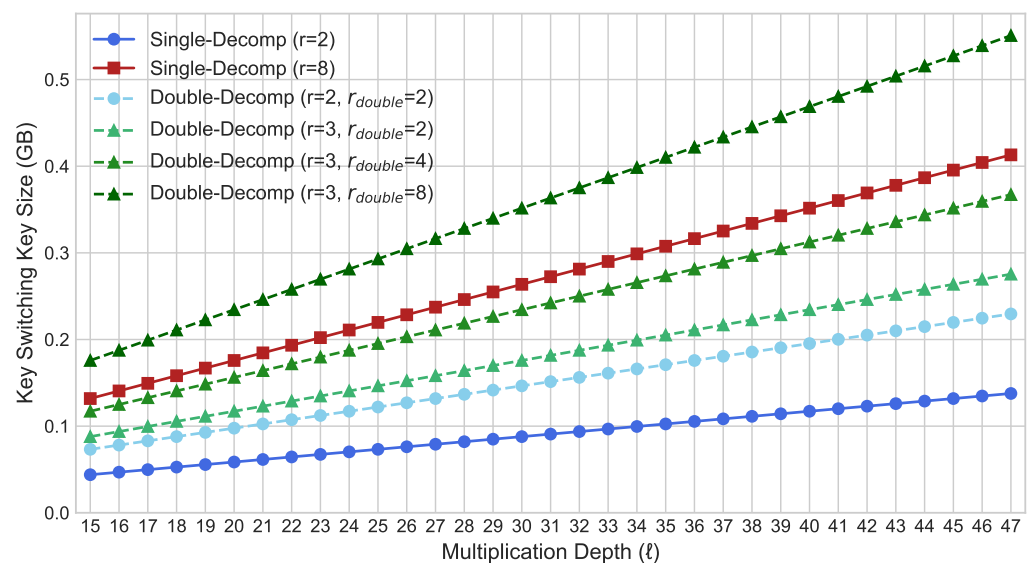


Figure 3. Key-switching key size growth vs. multiplication depth (ℓ), with $N = 2^{16}$ and each coefficient takes 8 bytes.

Table 5. Summary of maximum key sizes for different polynomial degrees N .

Polynomial Degree N	ℓ	Key Size (MB)	
		Max Single Decomposition	Max Double Decomposition
2^{12}	60	221	9071
2^{13}	60	443	18,143
2^{14}	60	885	36,285
2^{15}	60	1770	72,570
2^{16}	60	3540	145,140

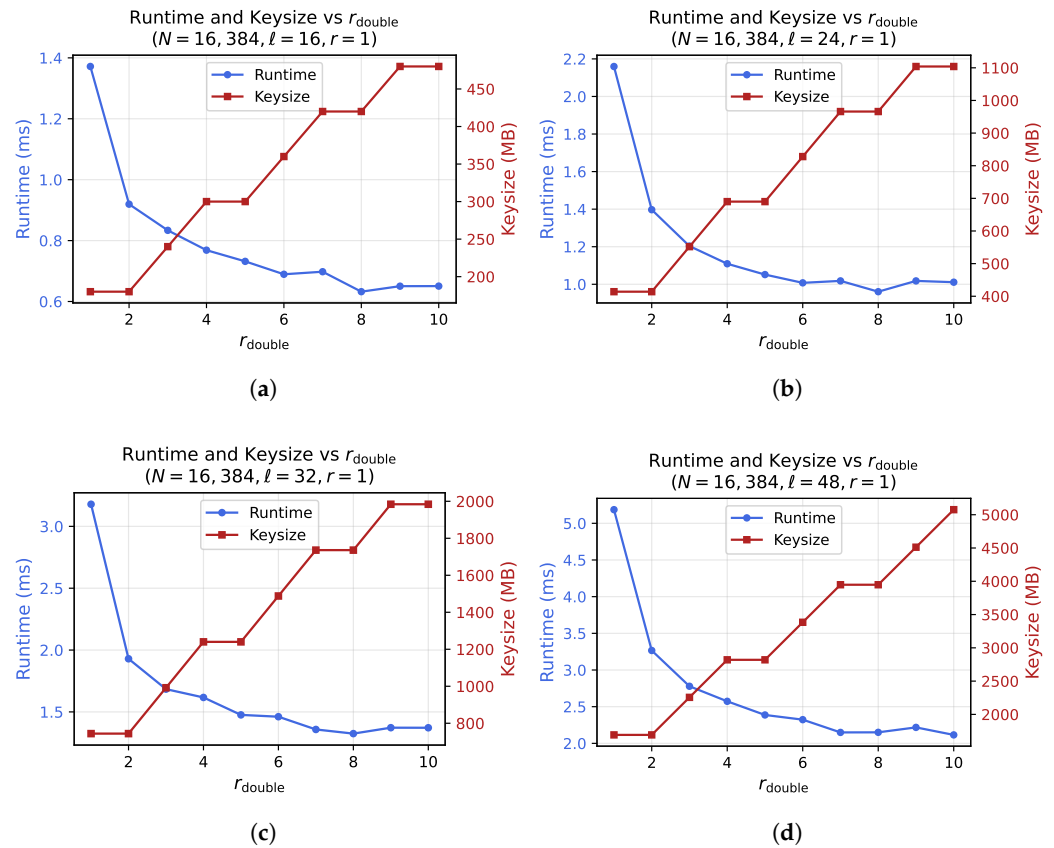


Figure 4. Runtime for one key-switching operation vs. key-switching key size trade-off for varying r_{double} , with $N = 16,384$. (a) $\ell = 16$. (b) $\ell = 24$. (c) $\ell = 32$. (d) $\ell = 48$.

7. Conclusions and Future Work

In this work, we addressed the growing need for high-performance and flexible GPU acceleration in FHE. We bridged a significant gap in the existing ecosystem by presenting the first unified, GPU-accelerated implementation of the KLSS key-switching algorithm for the BGV, BFV, and CKKS schemes.

Our extensive benchmarks confirmed that our CUDA-optimized implementation delivers significant speedups over both multi-core CPU baselines and prior state-of-the-art GPU implementations, reinforcing the critical role of GPUs in advancing practical FHE. More importantly, our work provides a detailed analysis of the trade-offs in key-switching design for GPU architectures. We experimentally validated the findings of Mono and Güneysu [46] that single decomposition offers superior latency, but we also demonstrated that this comes at the cost of reduced multiplicative depth. Our analysis highlights that double decomposition, despite its higher latency and memory footprint, remains an essential tool for depth-constrained applications where its efficient use of the modulus chain is paramount. This establishes a clear framework for practitioners: the choice is a strategic balance between latency, circuit depth, and the stringent memory limitations of GPU hardware.

Looking ahead, our work opens several promising avenues for future research. The unified framework and optimized kernels presented here are intentionally designed for extensibility. An immediate and natural extension is the integration of bootstrapping for all three schemes. Our proposed kernels can be directly leveraged for this purpose, as efficient key switching is a core component of the bootstrapping algorithm. Therefore, our work serves as a critical first step toward a complete, GPU-accelerated bootstrapping solution, which would enable computation of arbitrary depth. Further performance gains

could be realized by exploring more advanced CUDA features, optimizing memory transfer patterns, and investigating the use of specialized hardware units like Tensor Cores for polynomial arithmetic. Finally, extending our unified architecture to support other parallel platforms, such as FPGAs or other GPU vendors via APIs like SYCL, would further broaden the impact and accessibility of high-performance homomorphic encryption.

Author Contributions: Conceptualization, methodology, software, validation, S.J.; writing—original draft preparation, visualization, S.J.; writing—review and editing, supervision, R.C.C.C.; project administration, R.C.C.C.; funding acquisition, R.C.C.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported by CityUHK research grants 9239077 and 9239083.

Data Availability Statement: The data presented in this study are available on request from the corresponding author due to intellectual property considerations.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Notations

N	The polynomial ring degree, a power-of-two.
$\mathcal{R}_m$	The ring $\mathbb{Z}_m[X]/(X^N + 1)$.
Q_ℓ	The ciphertext modulus, $Q_\ell = \prod_{i=1}^{\ell} q_i$, composed of ℓ co-prime primes.
P	The key-switching extension modulus, $P = \prod_{j=1}^k P_j$, composed of k co-prime primes.
$\tilde{\ell}$	The total number of co-primes in the mod-up ring $\mathcal{R}_{Q_\ell P}$. $\tilde{\ell} = \ell + k$.
T	The double-decomposition modulus, $T = \prod_{i=1}^r T_i$, composed of r co-prime primes.
$\mathbf{s}_A, \mathbf{s}_B$	The old secret key ($\mathbf{s}_A$) and the new secret key ($\mathbf{s}_B$). For relinearization, $\mathbf{s}_B = \mathbf{s}_A^2$.
d	The number of decomposition groups for the first decomposition over q . The groups are denoted $q_{\bar{i}}$.
d_{double}	The number of decomposition groups for the second decomposition over QP . The groups are denoted $Q_{\bar{i}}$.
r	The RNS word width of the first decomposition.
r_{double}	The RNS word width of the second decomposition.
$\mathcal{D}, \mathcal{B}$	The first decomposition function and its reconstruction basis over Q_ℓ . For a polynomial $\mathbf{c}$, $\mathbf{c} = \langle \mathcal{D}(\mathbf{c}), \mathcal{B}(\mathbf{c}) \rangle$.
$\tilde{\mathcal{D}}, \tilde{\mathcal{B}}$	The second decomposition function and its reconstruction basis over $Q_\ell P$.
$\mathbf{c}_i$	The input ciphertext polynomial component to be switched, e.g., $\mathbf{c}_2$ from a multiplication result.
$\mathbf{ksk}$	The standard (single-decomposition) key-switching key.
$\mathbf{ksk}_{double}$	The double-decomposition key-switching key.

References

1. Zhang, Q.; Yang, L.T.; Chen, Z. Privacy Preserving Deep Computation Model on Cloud for Big Data Feature Learning. *IEEE Trans. Comput.* **2016**, *65*, 1351–1362. <https://doi.org/10.1109/TC.2015.2470255>.
2. Chabanne, H.; Wargny, A.; Milgram, J.; Morel, C.; Prouff, E. Privacy-preserving classification on deep neural network. *Cryptol. ePrint Arch.* **2017**. Available online: <https://ia.cr/2017/035> (accessed on 30 Oct 2025).
3. Jia, H.; Cai, D.; Huo, Z.; Wang, C.; Zhang, S.; Zhang, S.; Li, X.; Yang, S. Evaluation of Activation Functions in Convolutional Neural Networks for Image Classification Based on Homomorphic Encryption. In Proceedings of the 13th International Conference on Computer Engineering and Networks, Wuxi, China, 3–5 November 2023; Lecture Notes in Electrical Engineering; Springer Nature: Singapore, 2024; Volume 1127, pp. 343–355. https://doi.org/10.1007/978-981-99-9247-8_34.
4. Jiang, S.; Yang, H.; Xie, Q.; Ma, C.; Wang, S.; Xing, G. Lancelot: Towards Efficient and Privacy-Preserving Byzantine-Robust Federated Learning Within Fully Homomorphic Encryption. *arXiv* **2024**, arXiv:2408.06197. <https://doi.org/10.48550/arXiv.2408.06197>.

5. Asiri, M.; Khemakhem, M.A.; Alhebshi, R.M.; Alsulami, B.S.; Eassa, F.E. Decentralized Federated Learning for IoT Malware Detection at the Multi-Access Edge: A Two-Tier, Privacy-Preserving Design. *Future Internet* **2025**, *17*, 475. <https://doi.org/10.3390/fi17100475>.
6. Kim, M.; Lee, D.; Seo, J.; Song, Y. Accelerating HE operations from key decomposition technique. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 2024; Springer: Berlin/Heidelberg, Germany, 2023; pp. 70–92.
7. Brakerski, Z.; Gentry, C.; Vaikuntanathan, V. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory TOCT* **2014**, *6*, 309–325.
8. Fan, J.; Vercauteren, F. Somewhat practical fully homomorphic encryption. *Cryptol. ePrint Arch.* **2012**. Available online: <https://eprint.iacr.org/2012/144> (accessed on 30 Oct 2025).
9. Brakerski, Z.; Vaikuntanathan, V. Efficient fully homomorphic encryption from (standard) LWE. *SIAM J. Comput.* **2014**, *43*, 831–871.
10. Brakerski, Z.; Vaikuntanathan, V. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In Proceedings of the Annual Cryptology Conference, Santa Barbara, CA, USA, 14–18 August 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 505–524.
11. Cheon, J.H.; Kim, A.; Kim, M.; Song, Y. Homomorphic encryption for arithmetic of approximate numbers. In Proceedings of the Advances in Cryptology—ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, 3–7 December 2017; Springer: Berlin/Heidelberg, Germany, 2017; Proceedings, Part I 23, pp. 409–437.
12. Gentry, C. A Fully Homomorphic Encryption Scheme. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 2009.
13. Zvika, B.; Vinod, V. Lattice-based FHE as secure as PKE. In Proceedings of the 5th Conference on Innovations in Theoretical Computer Science, ITCS'14, New York, NY, USA, 12–14 January 2014; pp. 1–12. <https://doi.org/10.1145/2554797.2554799>.
14. Gentry, C.; Sahai, A.; Waters, B. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Proceedings of the Advances in Cryptology—CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 2013; Springer: Berlin/Heidelberg, Germany, 2013; Proceedings, Part I, pp. 75–92.
15. Naehrig, M.; Lauter, K.; Vaikuntanathan, V. Can homomorphic encryption be practical? In Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, Chicago, IL, USA, 21 October 2011; pp. 113–124.
16. Gentry, C.; Halevi, S.; Smart, N.P. Homomorphic Evaluation of the AES Circuit. *Cryptol. ePrint Arch.* **2012**. Available online: <https://eprint.iacr.org/2012/099> (accessed on 30 Oct 2025).
17. Aslett, L.; Esperança, P.; Holmes, C. Encrypted statistical machine learning: New privacy preserving methods. *arXiv* **2015**, arXiv:1508.06845.
18. Aono, Y.; Hayashi, T.; Wang, L.; Moriai, S. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Trans. Inf. Forensics Secur.* **2017**, *13*, 1333–1345.
19. Chen, H.; Cammarota, R.; Valencia, F.; Regazzoni, F.; Koushanfar, F. Ahec: End-to-end compiler framework for privacy-preserving machine learning acceleration. In Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC), Virtual, 20–24 July 2020; IEEE: New York, NY, USA, 2020; pp. 1–6.
20. Juvekar, C.; Vaikuntanathan, V.; Chandrakasan, A. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 1651–1669.
21. Ao, W.; Boddeti, V.N. AutoFHE: Automated Adaption of CNNs for Efficient Evaluation over FHE. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–18 August 2024; pp. 2173–2190.
22. Chen, H.; Laine, K.; Player, R. Simple encrypted arithmetic library-SEAL v2.1. In Proceedings of the Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, 7 April 2017; Springer: Berlin/Heidelberg, Germany, 2017; Revised Selected Papers 21, pp. 3–18.
23. Halevi, S.; Shoup, V. Design and implementation of a homomorphic-encryption library. *IBM Res.* **2013**, *6*, 8–36.
24. Al Badawi, A.; Bates, J.; Bergamaschi, F.; Cousins, D.B.; Erabelli, S.; Genise, N.; Halevi, S.; Hunt, H.; Kim, A.; Lee, Y.; et al. Openfhe: Open-source fully homomorphic encryption library. In Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022; pp. 53–63.
25. Kim, A.; Polyakov, Y.; Zucca, V. Revisiting Homomorphic Encryption Schemes for Finite Fields. In Proceedings of the Advances in Cryptology—ASIACRYPT 2021, Singapore, 6–10 December 2021; 2021; pp. 608–639. https://doi.org/10.1007/978-3-030-92078-4_21.
26. Han, K.; Ki, D. Better Bootstrapping for Approximate Homomorphic Encryption. *Cryptol. ePrint Arch.* **2019**. Available online: <https://eprint.iacr.org/2019/688> (accessed on 30 Oct 2025).
27. Zhou, L.; Huang, R.; Wang, B. Enhancing Multi-Key Fully Homomorphic Encryption with Efficient Key Switching and Batched Multi-Hop Computations. *Appl. Sci.* **2025**, *15*, 5771. <https://doi.org/10.3390/app15105771>.

28. Hwang, I.; Seo, J.; Song, Y. Optimizing HE operations via Level-aware Key-switching Framework. In Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023; pp. 59–67.
29. Wang, W.; Hu, Y.; Chen, L.; Huang, X.; Sunar, B. Accelerating fully homomorphic encryption using GPU. In Proceedings of the 2012 IEEE Conference on High Performance Extreme Computing, Waltham, MA, USA, 10–12 September 2012; IEEE: New York, NY, USA, 2012; pp. 1–5.
30. Wang, W.; Chen, Z.; Huang, X. Accelerating leveled fully homomorphic encryption using GPU. In Proceedings of the 2014 IEEE International Symposium on Circuits and Systems (ISCAS), Melbourne, Australia, 1–5 June 2014; IEEE: New York, NY, USA, 2014; pp. 2800–2803.
31. Dai, W.; Sunar, B. cuHE: A homomorphic encryption accelerator library. In Proceedings of the Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, 3–4 September 2015; pp. 169–186.
32. Al Badawi, A.; Veeravalli, B.; Mun, C.F.; Aung, K.M.M. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, 2018, 70–95.
33. Al Badawi, A.; Veeravalli, B.; Lin, J.; Xiao, N.; Kazuaki, M.; Mi, A.K.M. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Trans. Parallel Distrib. Syst.* **2020**, 32, 379–391.
34. Goey, J.Z.; Lee, W.K.; Goi, B.M.; Yap, W.S. Accelerating number theoretic transform in GPU platform for fully homomorphic encryption. *J. Supercomput.* **2021**, 77, 1455–1474.
35. Alves, P.G.M.; Ortiz, J.N.; Aranha, D.F. Faster homomorphic encryption over GPGPUs via hierarchical DGT. In Proceedings of the International Conference on Financial Cryptography and Data Security, Virtual, 1–5 March 2021; Springer: Berlin/Heidelberg, Germany, 2021; pp. 520–540. Creator (if available). Title of the Software, version, revision or edition (if available); comments (if available); Publisher: Place of Publication, Year.
36. vernamlab. cuFHE; GitHub: <https://github.com/vernamlab/cuFHE>, 2018
37. Xiao, Y.; Liu, F.H.; Ku, Y.T.; Ho, M.C.; Hsu, C.F.; Chang, M.C.; Hung, S.H.; Chen, W.C. GPU Acceleration for FHEW/TFHE Bootstrapping. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2025**, 2025, 314–339. <https://doi.org/10.46586/tches.v2025.i1.314-339>.
38. Shen, S.; Yang, H.; Liu, Z.; Liu, Y.; Lu, X.; Dai, W.; Zhou, L.; Zhao, Y.; Cheung, R.C.C. VeloFHE: GPU Acceleration for FHEW and TFHE Bootstrapping. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2025**, 2025, 81–114. <https://doi.org/10.46586/tches.v2025.i3.81-114>.
39. Jin, S.; Shen, S.; Yang, H.; Chen, D.; Dai, W.; Cheung, R.C.C. CuFDFB: Fast and Private Computation on Non-Linear Functions Using FHE. *Cryptol. ePrint Arch.* **2025**, 14, 1–13. Available online: <https://eprint.iacr.org/2025/1096> (accessed on 30 Oct 2025).
40. Jung, W.; Kim, S.; Ahn, J.H.; Cheon, J.H.; Lee, Y. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2021**, 2021, 114–148. <https://doi.org/10.46586/tches.v2021.i4.114-148>.
41. Yang, H.; Shen, S.; Dai, W.; Zhou, L.; Liu, Z.; Zhao, Y. Phantom: A CUDA-accelerated Word-Wise Homomorphic Encryption Library. *IEEE Trans. Dependable Secur. Comput.* **2024**, 21, 4895–4906.
42. Ozcan, A.S.; Savas, E. HEonGPU: A GPU-Based Fully Homomorphic Encryption Library 1.0. *Cryptol. ePrint Arch.* **2024**. Available online: <https://eprint.iacr.org/2024/1543> (accessed on 30 Oct 2025).
43. Fan, S.; Wang, Z.; Xu, W.; Hou, R.; Meng, D.; Zhang, M. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 25 February–1 March 2023; pp. 922–934, ISSN: 2378-203X. <https://doi.org/10.1109/HPCA56546.2023.10071017>.
44. Fan, G.; Zhang, M.; Zheng, F.; Fan, S.; Zhou, T.; Deng, X.; Tang, W.; Kong, L.; Song, Y.; Yan, S. WarpDrive: GPU-Based Fully Homomorphic Encryption Acceleration Leveraging Tensor and CUDA Cores. In Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA), Las Vegas, NV, USA, 1–5 March 2025; pp. 1187–1200, ISSN: 2378-203X. <https://doi.org/10.1109/HPCA61900.2025.00091>.
45. Kim, J.; Choi, W.; Ahn, J.H. Cheddar: A Swift Fully Homomorphic Encryption Library for CUDA GPUs. *arXiv* **2024**, arXiv:2407.13055. <https://doi.org/10.48550/arXiv.2407.13055>.
46. Mono, J.; Güneysu, T. A New Perspective on Key Switching for BGV-like Schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2025**, 2025, 763–794. <https://doi.org/10.46586/tches.v2025.i4.763-794>.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.