IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 3, pp. 81–114. DOI:10.46586/tches.v2025.i3.81-114

VeloFHE: GPU Acceleration for FHEW and TFHE Bootstrapping

Shiyu Shen[1], Hao Yang[1], Zhe Liu[2(✉)], Ying Liu[3,4], Xianhui Lu[3,4], Wangchen Dai[5], Lu Zhou[6], Yunlei Zhao[7] and Ray C. C. Cheung[1(✉)]

[1] City University of Hong Kong, Hong Kong, China,
crypto@sher1e.dev,crypto@d4rk.dev,r.cheung@cityu.edu.hk
[2] Zhejiang Lab, Hangzhou, China, zhe.liu@nuaa.edu.cn
[3] Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[4] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[5] Sun Yat-sen University, Shenzhen, China
[6] Nanjing University of Aeronautics and Astronautics, Nanjing, China,
[7] Fudan University, Shanghai, China

Abstract. Bit-wise Fully Homomorphic Encryption schemes like FHEW and TFHE offer efficient functional bootstrapping, enabling concurrent function evaluation and noise reduction. While advantageous for secure computations, these schemes suffer from high data expansion, posing significant performance challenges in practical applications due to massive ciphertexts. To address these issues, we propose VeloFHE, a CUDA-accelerated design to enhance the efficiency of FHEW and TFHE schemes on GPUs. We develop a novel hybrid four-step Number Theoretic Transform (NTT) approach for fast polynomial multiplication. By decomposing large-scale NTTs into highly parallelizable submodules, incorporating cyclic and negacyclic convolutions, and introducing several memory-oriented optimizations, we significantly reduce both the computational complexity and memory requirements. For blind rotation, besides the gadget decomposition approach, we also apply a recent proposed modulus raising technique to both schemes to alleviate memory pressure. We further optimize it by refining computational flow to reduce noise from scaling and maintain accumulator compatibility. For key switching, we address input-output parallelism mismatches, and offloading suitable computations to the CPU, effectively hiding latency through asynchronous execution. Additionally, we explore batching in bootstrapping, developing a general framework that accommodates both schemes with either gadget decomposition or modulus raising method.

Our experimental results demonstrate significant performance improvements. The proposed NTT implementation shows over 35% improvement compared to recent GPU implementations. On an RTX 4090 GPU, we achieve speedups of 371.86× and 390.44× for FHEW and TFHE gate bootstrapping, respectively, compared to OpenFHE running on a 48-thread CPU at a 128-bit security level. The corresponding throughputs are 7,007 and 11,378 operations per second. Furthermore, relative to the state-of-the-art GPU implementation [XLK+25], our approach provides speedups of 2.56×, 2.24×, and 2.33× for TFHE gate bootstrapping, homomorphic evaluation of arbitrary functions, and homomorphic flooring operation, respectively. Our VeloFHE surpasses some current hardware designs, offering an effective solution for more practical and efficient privacy-preserving computations.

Keywords: Fully Homomorphic Encryption · Bootstrapping · FHEW · TFHE · GPU acceleration

Received: 2025-01-15 Accepted: 2025-03-15 Published: 2025-06-05

1 Introduction

Homomorphic Encryption (HE) enables computations on ciphertexts without access to the secret key, making it crucial for privacy protection in scenarios like secure outsourcing. Currently, it has become a fundamental component of privacy-preserving applications. Despite its promise, HE schemes often have limited computational depth. In large-scale applications, bootstrapping is necessary to allow further computation and correct decryption. The concept of bootstrapping is introduced by Gentry [Gen09], which makes construction of fully homomorphic encryption (FHE) scheme possible. Despite several subsequent improvements, it still remains as a significant performance bottleneck.

Modern FHE schemes are generally categorized as word-wise FHE (e.g., BGV [BGV12], BFV [Bra12, FV12], and CKKS [CKKS17]) and bit-wise FHE (e.g., FHEW [DM15] and TFHE [CGGI16, CGGI20]). Word-wise schemes support larger plaintext spaces and efficient data packing, reducing the amortized bootstrapping time but results in high overall latency. In contrast, bit-wise schemes operate on a few bits per ciphertext but achieve faster bootstrapping, often within seconds. These schemes are suitable for evaluating non-polynomial functions and offer advantages in specific applications.

FHEW and TFHE are among the most efficient bit-wise FHE methods. FHEW is based on the Alperin-Sheriff and Peikert (AP) method [AP14] with an efficient instantiation in the ring setting for bootstrapping. TFHE further enhances it by leveraging the Gama, Izabachène, Nguyen, and Xie (GINX) method [GINX16, CGGI20]. However, certain limitations hinder their broader application. In practical scenarios, the data to be encrypted is typically large, but bit-wise schemes can encrypt only a few bits per ciphertext. This leads to significant ciphertext expansion and a substantial number of ciphertexts to process, resulting in increased computational, IO, and memory latency.

Hardware acceleration like GPUs is often proposed to address performance challenges. GPUs are powerful parallel computing platforms suitable for accelerating FHE schemes due to their inherent parallelism. The lattice-based structure offers natural opportunities for parallelizing, and many applications involve independent homomorphic operations ideal for batching. However, accelerating bit-wise FHEs on GPUs presents challenges. While GPUs excel at compute-intensive tasks, the large data volume in bit-wise FHE schemes makes computation memory-bound. Excessive data access to GPU global memory incurs high latency. Using faster memory like shared memory or registers can alleviate it, but balancing this with high hardware occupancy is complex due to the substantial data size.

1.1 Our Contributions

In this work, we introduce VeloFHE, a CUDA-accelerated implementation of the FHEW and TFHE homomorphic encryption schemes. By tackling computational bottlenecks and employing memory-optimized strategies, we significantly enhance the performance of these schemes on GPUs.

One of the most computationally intensive operations in these schemes is polynomial multiplication. For GPU-based implementations, it is beneficial to decompose the computational task into smaller, embarrassingly parallel submodules to effectively manage both computation and memory overhead. Drawing inspiration from this principle, we leverage the four-step Number Theoretic Transform (NTT) framework to decompose large-scale NTT computations, thereby making them more suitable for GPU architectures. We further propose a warp-based design that offers several advantages. This makes computations more focused on relevant domains, obviates the need for synchronization, and enables more scalable instantiation of thread blocks. However, directly porting the four-step NTT to GPUs poses several challenges, including excessive memory access, inefficient data-access patterns, and frequent matrix transpositions. To overcome these limitations, we propose a range of theoretical and implementation-level optimizations. In particular,

our hybrid four-step NTT approach supports both cyclic and negacyclic convolutions, enables implicit transposition, and ensures uncoalesced, conflict-free memory access. These features collectively improve both computational and memory efficiency. Compared to existing approaches, this design reduces the reliance on GPU global memory access and synchronization overhead. It also supports broader operation fusion, paving the way for additional performance improvements.

Blind rotation and key switching are critical components in both AP- and GINX-type bootstrapping. In the literature, two methods have been applied for the external product used in blind rotation. The first is the gadget decomposition approach [DM15, CGGI16], which remains the mainstream method but incurs high computational and memory overhead. The second is the modulus raising approach [LLL+24], recently introduced for TFHE. Although it is more memory-efficient, it faces challenges related to noise control and limited parameter flexibility. We include both approaches in VeloFHE. Specifically, for the modulus raising approach, we further extend it to FHEW, and propose an optimized computation flow that reduces the noise introduced during processing. Our compact NTT design also allows the entire accumulation phase to be fused into a single kernel, which minimizes memory access latency. For key switching, we propose a dimension-aware strategy to effectively handle kernel input-output mismatches, solving performance bottlenecks. Additionally, we partition the workload and delegate specific computations to the CPU, preventing single-threaded execution and idle waiting. By asynchronously execution, we effectively hide the CPU execution latency. Building upon these optimizations, we further investigate batching in bootstrapping. We develop a general framework that accommodates both AP and GINX methods with either gadget decomposition or modulus raising. To address noncontiguous data transfers during workload partitioning, we introduce a memory pooling mechanism, optimizing memory layout for efficient data transfer.

Owing to the aforementioned optimizations, we achieve substantial performance improvements. Our hybrid four-step NTT implementation delivers more than a 35% performance increase over the latest GPU implementations. Under the standard `STD128` parameter set, we achieve speedups of 371.86× and 390.44× for gate bootstrapping in the two schemes compared to OpenFHE [ope24] running on a 48-thread CPU, yielding throughputs of 7,007 and 11,378 operations per second, respectively. These results exceed the performance of certain cutting-edge hardware implementations. Relative to the state-of-the-art GPU implementations [XLK+25], we achieve speedups of 2.56×, 2.24×, and 2.33× for TFHE gate bootstrapping, homomorphic evaluation of arbitrary functions, and homomorphic flooring operation, respectively. Furthermore, to the best of our knowledge, this work is the first to report performance results for bootstrapping with the recently introduced modulus raising technique on GPUs.

In summary, our contributions are as follows:

- A novel hybrid four-step NTT design that is highly parallelizable, reduces memory overhead and synchronization costs, and scales efficiently across varying parameters and block configurations.
- Optimized blind rotation and key switching implementations for both AP type and GINX type bootstrapping. This includes a modulus raising technique for blind rotation with better noise control ability, and new fusion strategies that minimize memory transfers.
- VeloFHE, an integrated CUDA-accelerated framework that consolidates these innovations for FHEW and TFHE, delivering substantial speedups over existing CPU and GPU solutions.

1.2 Related Works

FHEW and TFHE are among the leading methods for practical bootstrapping in FHE, which enable look-up tables (LUTs) evaluation during bootstrapping. Below, we present

an overview of existing researches and provide comparative analysis with our work.

1) Theoretical Advancements. Since their introduction, FHEW and TFHE have been further enhanced on functionality, enable features like packing [CGGI17], supporting extended secret distributions [MP21], enhancing homomorphic evaluation capacity of functions [CIM19, GBA21], and broadening the input domain [BDF18, KS23]. However, these schemes still face performance challenges due to large ciphertext sizes and gigabyte-scale bootstrapping keys. Various efforts aim to reduce the complexity of blind rotation, for instance, by reconstructing blind rotation using the NTRU assumption [BIP+22, XZD+23] or Galois automorphisms [LMK+23], and by combining gadget decomposition with modulus raising [LLL+24]. Our work adopts the two scheme variants proposed in [MP21] and proposes refined bootstrapping with modulus raising technique. Compared to [LLL+24], our approach generalizes modulus raising to FHEW, and provides enhanced noise control. Moreover, our implementation can be easily applied to other aforementioned variants.

2) Software Implementations. Several open-source libraries implement these schemes, including FHEW [fhe24], Concrete [CJL+20], TFHE-rs [Zam22], and OpenFHE [ope24]. Among these, many solutions focus on binary-key TFHE, while OpenFHE [ope24] implements a ternary-key variant [MP21] that more closely adheres to the HE Standard [ACC+21]. Our VeloFHE supports both binary and ternary secret-key distributions for TFHE, and it remains compatible with OpenFHE to further improve performance. The TFHEpp [Mat20] and MOSFHET library [GBA24] accelerates computations using Intel AVX2/FMA/AVX512 instructions. GPUs frequently offer superior performance due to their ability to exploit the inherent parallelism of operations and efficiently batch-process multiple operations concurrently.

3) GPU Acceleration. GPU acceleration has been intensively studied for word-wise HE schemes like BGV, BFV, and CKKS [BVMA18, JKA+21, BPA+21, TÖA+22, SYL+23, FWX+23, YSD+24, SYD+24, ÖS24], and supported by open-source libraries such as Phantom [pha24] and HEonGPU [heo24]. However, bit-wise FHE GPU implementations remain limited and often focus on specific parameters and operations. For example, the cuFHE [cuf24] and nuFHE library [nuf] only support fixed paramters and gate bootstrapping. Other studies have targeted narrow aspects, such as only accelerating the blind rotation [LLCP15], FHEW-based homomorphic full adder [LGZ+19], or TFHE-based algebraic circuits [MAM20]. A recent parallel work [XLK+25] also accelerates FHEW and TFHE on GPUs, reporting that using Fast Fourier Transform (FFT) is more efficient than NTT in these schemes. Additionally, they develop a multi-block bootstrapping through CUDA cooperative kernel technique, enabling flexible adjustments to the number of thread blocks. In contrast, our implementation focuses on NTT with general parameter settings, while our warp-based design also offers flexible thread block configurations within a single block. Unlike FFT, NTT does not suffer from precision issues that could impact the decryption failure probability. This ensures that the security level is not compromised by decryption failure attacks in certain security-sensitive scenarios. Furthermore, our implementation demonstrates superior performance compared to [XLK+25]. Notably, our four-step NTT design can also be adapted to FFT, offering the potential for additional performance improvements in future developments.

4) Hardware Accelerators. FPGA and ASIC designs have been proposed for BGV [SFK+21, GBP+23], BFV [RJV+18, RTJ+19, RCK+21], CKKS [RLPD20, KLK+22, KKK+22, AdCY+23, KKC+23, AMK+23, MAK+23], FHEW [BBTV23], and TFHE [JLJ22, YKP22, BDTV23, PPC+23, PPK24, ZCL+24, MBS24], as well as solutions that support both CKKS and TFHE [DFH+24, ZNW+24]. Among these works, REED [AMK+23], OFHE [ZCL+24], and HOGE [MBS24] utilize four-step NTT/FFT, but there are differences compared to our approach. REED [AMK+23] serializes the second sub-NTT phase to address memory conflicts. OFHE [ZCL+24] fixes the sub-NTT size to 64, requiring padding of polynomials to match this size. HOGE [MBS24] employs Goldilocks prime that enable

modular reduction through bit masking. In contrast, our method enables parallelization of both sub-NTT phases and is designed for more general cases of polynomial sizes and moduli. GPU implementations offer greater flexibility since FHE schemes are still evolving and parameters are not fixed. While optimization goals on both platforms are similar, our approach introduces novel techniques for achieving these optimizations on GPUs. Additionally, we achieve same performance level comparable to some FPGA and ASIC designs, highlighting its effectiveness.

5) Privacy-Preserving Applications. FHEW and TFHE have been widely adopted in privacy-preserving applications, such as neural network inference [BMMP18, CJP21, TCBS23a] and AES circuit evaluation [TCBS23b, WWL+24]. Our proposed GPU-accelerated approach, which speeds up the basic building blocks of such applications, has the potential to further improve their performance. We have demonstrated the efficiency of fundamental operations, extending these optimizations to full-fledged privacy-preserving systems is a promising future direction.

2 Preliminaries

2.1 Notations

Let $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{Z}_q$ denote the set of integers, rational numbers, and integers modulo q, respectively. For N a power of 2, define the $2N$-th cyclotomic ring as $\mathcal{R}_{N,Q} = \mathbb{Z}_Q[X]/(X^N+1)$. For $x \in \mathbb{Q}$, $\lfloor x \rfloor$, $\lceil x \rceil$, and $\lfloor x \rceil$ denote rounding to the lower, higher, and closest integer, respectively. For integer a, $[a]_q$ refers to a mod q. $a \mid n$ denotes n is divisible by a. We use bold lowercase letters to denote ring elements (e.g., $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i X^i$) and vectors (e.g., $\mathbf{v}$), and bold uppercase letters for matrices (e.g., $\mathbf{A}$). $\mathbf{A}^\top$ denotes matrix transpose. The hat symbol indicates elements in the frequency domain. For a base $B_\mathbf{g}$, define $d_\mathbf{g} = \lfloor \log_{B_\mathbf{g}} Q \rceil + 1$. The gadget decomposition of $\boldsymbol{a} \in \mathcal{R}_Q$ is $g^{-1}(\boldsymbol{a}) = ([\boldsymbol{a}]_{B_\mathbf{g}}, [\boldsymbol{a}/B_\mathbf{g}]_{B_\mathbf{g}}, \ldots, [\boldsymbol{a}/B_\mathbf{g}^{d_\mathbf{g}-1}]_{B_\mathbf{g}}) \in \mathcal{R}_{B_\mathbf{g}}^{d_\mathbf{g}}$. The corresponding expansion function is $g = (B_\mathbf{g}^0, B_\mathbf{g}, \ldots, B_\mathbf{g}^{d_\mathbf{g}-1}) \in \mathcal{R}_Q^{d_\mathbf{g}}$.

2.2 Cryptosystems and Arithmetics

The Learning with Errors (LWE) problem, introduced by Regev [Reg05], enhances encryption security by incorporating noise. Given a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$, the LWE encryption of a plaintext $m \in \mathbb{Z}_q$ is denoted as $\mathsf{LWE}_\mathbf{s}^{n,q}(m)$. The corresponding ciphertext is $\mathbf{ct} = (\mathbf{a}, b) = (\mathbf{a}, \mathbf{a}^\top\mathbf{s} + m + e)$, where $\mathbf{a} \in \mathbb{Z}_q^n$ is chosen uniformly at random, and e is sampled from χ_e. Decryption computes $m' = b - \mathbf{a}^\top\mathbf{s}$ to recover the plaintext.

The Ring LWE (RLWE) problem extends LWE to polynomial rings. An RLWE encryption of $\boldsymbol{m} \in \mathcal{R}_{N,Q}$ under key $\boldsymbol{s} \in \mathcal{R}_N$ is $\mathbf{ct} = (\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a}, \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{m} + \boldsymbol{e})$, with $\boldsymbol{a} \in \mathcal{R}_{N,Q}$ uniformly random and $\boldsymbol{e}$ sampled from $\chi_{\boldsymbol{e}}$. For simplicity, we represent $\boldsymbol{s}$ as a vector $\mathbf{s}$ with components $\mathbf{s}[i] = s_i$. This encryption process is denoted as $\mathsf{RLWE}_\mathbf{s}^{N,Q}(\boldsymbol{m})$, and decryption is achieved by computing $\boldsymbol{m'} = \boldsymbol{b} - \boldsymbol{a} \cdot \boldsymbol{s}$, recovering $\boldsymbol{m}$ by $\boldsymbol{m'} \approx \boldsymbol{m}$.

To support complex arithmetic operations of ciphertexts, we introduce the extended RLWE encryption $\mathsf{RLWE}'_\mathbf{s}(\boldsymbol{m}) = (\mathsf{RLWE}_\mathbf{s}(\boldsymbol{m}), \mathsf{RLWE}_\mathbf{s}(B\boldsymbol{m}), \ldots, \mathsf{RLWE}_\mathbf{s}(B^{k-1}\boldsymbol{m}))$, where $k = \log_B Q$ and B is a base for element decomposition. Then, we adopt the ring variant of the GSW scheme [GSW13, MP21], denoted by $\mathsf{RGSW}_\mathbf{s}(\boldsymbol{m}) = (\mathsf{RLWE}'_\mathbf{s}(-\boldsymbol{s}\cdot\boldsymbol{m}), \mathsf{RLWE}'_\mathbf{s}(\boldsymbol{m}))$. These cryptosystems support various homomorphic operations on RLWE ciphertexts:

- **Multiplication by an Arbitrary Plaintext ($\odot$).** When the plaintext $\boldsymbol{d}$ has a large norm, direct multiplication can cause excessive noise growth. To mitigate this, we decompose $\boldsymbol{d}$ into base B digits by $\boldsymbol{d} = \sum_{i=0}^{k-1} B^i \boldsymbol{d}_i$. Given an extended RLWE ciphertext $\mathbf{ct} = (\mathbf{c}_0, \ldots, \mathbf{c}_{k-1}) = \mathsf{RLWE}'_\mathbf{s}(\boldsymbol{m})$, the multiplication is computed as $\boldsymbol{d} \odot \mathbf{ct} = \sum_i \boldsymbol{d}_i \cdot \mathbf{c}_i$. This results in a ciphertext encrypting $\boldsymbol{d} \cdot \boldsymbol{m}$ with controlled

noise growth. An extended product can be obtained via $\boldsymbol{d} \odot' \mathbf{ct} = (\boldsymbol{d} \odot \mathbf{ct}, (B\boldsymbol{d}) \odot \mathbf{ct}, \ldots, (B^{k-1}\boldsymbol{d}) \odot \mathbf{ct})$.

- **External Product ($\diamond$).** This operation enables multiplication between RLWE and RGSW ciphertexts. Given $\mathbf{ct} = (\boldsymbol{a}, \boldsymbol{b}) = \mathsf{RLWE}_{\mathbf{s}}(\boldsymbol{m})$ and $\mathbf{ct}' = (\boldsymbol{c}_0, \boldsymbol{c}_1) = \mathsf{RGSW}_{\mathbf{s}}(\boldsymbol{m}')$, the external product is computed as $\mathbf{ct} \diamond \mathbf{ct}' = \boldsymbol{a} \odot \boldsymbol{c}_0 + \boldsymbol{b} \odot \boldsymbol{c}_1$, resulting in an RLWE ciphertext encrypting $\boldsymbol{m} \cdot \boldsymbol{m}'$. This operation extends naturally to scenarios of $\mathsf{RLWE}' \times \mathsf{RGSW}$ and $\mathsf{RGSW} \times \mathsf{RGSW}$ through component-wise computation.

2.3 Bit-Wise HE Schemes and Building Blocks

FHEW and TFHE are bit-wise FHE schemes that operate on bits, enabling homomorphic evaluation of Boolean functions such as NAND and XOR. Constructed based on the (R)LWE problem, these schemes feature fast bootstrapping with millisecond range and can evaluate LUTs during bootstrapping. They are particularly suitable for homomorphic evaluation of non-polynomial functions in privacy-preserving computations.

In this work, we focus on the schemes introduced in [MP21], which employ uniform ternary secret keys over $\{-1, 0, 1\}$ for TFHE. The ciphertext space is $\mathbb{Z}_q^n \times \mathbb{Z}_q$, and the ring used in RGSW encryption is $\mathcal{R}_{N,Q} = \mathbb{Z}_Q[X]/(X^N + 1)$. Common choices for N are $N \in \{2^{10}, 2^{11}, 2^{12}\}$, with $q \mid 2N$ to allow RGSW encryption of LWE samples in the exponent, and $Q \mid (2N - 1)$ to facilitate fast polynomial multiplication. Below, we detail the building blocks of these schemes.

Blind Rotation. Blind rotation enables homomorphic decryption of an LWE ciphertext using an RGSW cryptosystem. For an LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, we use $\mathsf{RGSW}_{\mathbf{s}'}^{N,Q}$, with $q \mid 2N$, to encrypt each component s_i of the secret key $\mathbf{s}$. Starting with a homomorphic accumulator $\mathsf{acc} = (0, X^b \cdot tv)$, where tv is a test vector, blind rotation updates the accumulator n times to get $\mathbf{ct}_B \in \mathsf{RLWE}_{\mathbf{s}'}^{N,Q}(X^{b - \sum_{i=0}^{n-1} a_i \cdot s_i} \cdot tv)$. This rotates the term corresponding to m to the polynomial's constant term without revealing m.

There are two primary methods for implementing blind rotation: the AP [AP14, DM15] and the GINX method [GINX16, CGGI20]. The AP method requires decomposition of the ciphertext and can be applied to secrets of any size, but it results in larger evaluation keys. The GINX method relies on binary secrets, providing performance gains through a CMUX operation. However, it is computationally costly for other secret distributions.

Sample Extraction. Sample extraction allows for the extraction of an LWE ciphertext corresponding to a specific coefficient from an RLWE ciphertext. Given an RLWE ciphertext $\mathbf{ct} = (\boldsymbol{a}, \boldsymbol{b}) = \mathsf{RLWE}_{\mathbf{s}}^{N,Q}(\boldsymbol{m})$, where $\boldsymbol{m} = \sum_{i=0}^{N-1} m_i X^i$, the operation outputs an LWE encryption of the coefficient m_i at index i. This is particularly useful for extracting the constant term after performing a blind rotation. In the case of $i = 0$, the resulting LWE ciphertext is $(a_0, -a_{N-1}, \ldots, -a_1, b_0) = \mathsf{LWE}_{\mathbf{s}}^{N,Q}(m_0)$.

Key Switching. Key switching changes the underlying secret key of a ciphertext to a different secret key while preserving the encrypted message, using a switching key. We describe a well-known method for LWE ciphertexts. Formally, given an LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{N,Q_{\mathtt{ks}}}(m)$ and a switching key $\mathbf{ksk}_{\mathbf{s} \to \mathbf{s}'} = (\mathbf{ksk}_0, \mathbf{ksk}_1)$, the operation outputs a new ciphertext $\mathbf{ct}' = (\mathbf{a}', b') \in \mathsf{LWE}_{\mathbf{s}'}^{n,Q_{\mathtt{ks}}}(m)$. The process involves two phases: generating the key-switching key and performing the key switching.

- *Key-Switching Key Generation.* To transition from the secret key $\mathbf{s}$ to $\mathbf{s}'$, the switching key is computed as $\mathbf{ksk}_{\mathbf{s} \to \mathbf{s}'} = \{\mathbf{ksk}_{i,j,v}\}$, where $\mathbf{ksk}_{i,j,v} \in \mathsf{LWE}_{\mathbf{s}}^{n,Q_{\mathtt{ks}}}(v s_i' B_{\mathtt{ks}}^j)$, with $v \in [0, B_{\mathtt{ks}})$, $i \in [0, N)$, $j \in [0, d_{\mathtt{ks}})$, $B_{\mathtt{ks}}$ being the base for decomposition, and $d_{\mathtt{ks}} = \lceil \log_{B_{\mathtt{ks}}} Q_{\mathtt{ks}} \rceil$ the number of digits.

Algorithm 1 Boostrapping in FHEW-like schemes

Input: LWE ciphertext $\mathbf{ct} \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, bootstrapping key $\mathbf{bsk}$, key-switching key $\mathbf{ksk}_{\mathbf{s}'\to\mathbf{s}}$, test vector tv embedding a LUT f.
Output: LWE ciphertext $\mathbf{ct}' \in \mathsf{RLWE}_{\mathbf{s}}^{n,q}(f(m))$

Set $\mathbf{acc} = (\mathbf{0}, X^b \cdot tv) \in \mathcal{R}_Q^2$
$\mathbf{ct}_B \leftarrow \mathsf{BlindRotate}(\mathbf{acc}, \mathbf{ct}, \mathbf{bsk}, tv)$ ▷ $\mathbf{ct}_B \in \mathsf{RLWE}_{\mathbf{s}'}^{N,Q}(f(m))$
$\mathbf{ct}_E \leftarrow \mathsf{SampleExtract}(\mathbf{ct}_B)$ ▷ Extract LWE ciphertext $\mathbf{ct}_E \in \mathsf{LWE}_{\mathbf{s}'}^{N,Q}(f(m))$
$\mathbf{ct}_M \leftarrow \mathsf{ModSwitch}(\mathbf{ct}_E, Q, Q_{\mathtt{ks}})$ ▷ Switch from Q to a temporary modulus $Q_{\mathtt{ks}}$
$\mathbf{ct}_K \leftarrow \mathsf{KeySwitch}(\mathbf{ct}_M, \mathbf{ksk}_{\mathbf{s}'\to\mathbf{s}})$ ▷ $\mathbf{ct}_K \in \mathsf{LWE}_{\mathbf{s}}^{n,Q_{\mathtt{ks}}}(f(m))$
$\mathbf{ct}' \leftarrow \mathsf{ModSwitch}(\mathbf{ct}_K, Q_{\mathtt{ks}}, q)$ ▷ Switch from Q_K to modulus q
return $\mathbf{ct}'$

- *Key Switching.* Given the ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{N,Q_{\mathtt{ks}}}(m)$ and the key-switching key $\mathbf{ksk}_{\mathbf{s}\to\mathbf{s}'}$, the operation first decomposes each element of $\mathbf{a}$ as $a_i = \sum_j a_{i,j} B_{\mathtt{ks}}^j$, and then computes $\mathbf{ct}' = (\mathbf{0}, b) - \sum_{i,j} \mathbf{ksk}_{i,j,a_{i,j}} \bmod Q_{\mathtt{ks}}$. The result is a new LWE ciphertext encrypting the same message m under the new secret key $\mathbf{s}'$.

Modulus Switching. Modulus switching changes the modulus of a ciphertext while preserving the encrypted plaintext. Given an LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{n,Q}(m)$, the operation generates a new ciphertext $\mathbf{ct}' = (\mathbf{a}', b') \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$ under a different modulus q by computing $\mathbf{ct}' = (\lfloor \frac{q}{Q} \cdot \mathbf{a} \rceil, \lfloor \frac{q}{Q} \cdot b \rceil)$. This technique allows compatibility to manage ciphertexts encrypted under different moduli, facilitating more flexible computations.

2.4 Bootstrapping

In the LWE cryptosystem, noise accumulates during homomorphic computations, which can eventually hinder correct decryption. Bootstrapping is a technique used to refresh ciphertexts, reducing noise and restoring decryption accuracy. There are two modes of bootstrapping, i.e., fully homomorphic evaluation and leveled homomorphic evaluation. The former offers lower latency by refreshing ciphertexts more frequently, while the latter allows for the computation of larger logic circuits before noise reduction is necessary.

The fully homomorphic evaluation mode employs functional bootstrapping. Given an LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, the method encrypts the secret key $\mathbf{s}$ using another secret key $\mathbf{s}'$ under the RGSW cryptosystem to create a bootstrapping key. Homomorphic decryption is then performed to refresh the ciphertext, as detailed in Algorithm 1. The process begins with the $\mathsf{BlindRotate}$ operation to homomorphically evaluate decryption. A look-up table $f : \mathbb{Z}_t \to \mathbb{Z}_t$ is embedded into a test vector by defining the polynomial coefficients as $tv = \sum_{i=0}^{N-1} \frac{Q}{t} \cdot f(\lfloor \frac{t}{q} \cdot i \rceil) \cdot X^i$. This transforms the accumulator into an RLWE encryption of $f(m)$ under the new secret key $\mathbf{s}'$, polynomial size N, and modulus Q, where $f(m)$ appears in the constant term. It is important to note that FHEW and TFHE utilize different blind rotation methods, AP and GINX, respectively. Following this, the $\mathsf{SampleExtract}$ operation extracts an LWE ciphertext from the RLWE ciphertext. The modulus is temporarily switched to $Q_{\mathtt{ks}}$ to perform key switching, changing the secret key and dimension to $\mathbf{s}$ and n, respectively. Finally, the modulus is reverted to the original value q.

Gate bootstrapping is a specific case of functional bootstrapping that evaluates a one-bit LUT. In this context, binary values $\{0, 1\}$ are represented by $\{-\frac{1}{4}, \frac{1}{4}\}$. For a one-bit message $m \in \{0, 1\}$, the encryption is given by $\mathsf{LWE}_{\mathbf{s}}^{n,q}(\frac{q}{4} \cdot m)$. Thus, adding two such ciphertexts results in $\mathbf{ct}_0 + \mathbf{ct}_1 = \mathsf{LWE}_{\mathbf{s}}^{n,q}(\frac{q}{4} \cdot (m_0 + m_1))$. Based on this, a function f can be designed to map the value range appropriately while satisfying the negacyclic property required for correct decryption. By choosing different map, various logic gates can be implemented. For instance, the NAND gate is computed as $\mathsf{Bootstrap}((\mathbf{0}, \frac{5}{8}) - \mathbf{ct}_0 - \mathbf{ct}_1)$.

2.5 Number Theoretic Transform

The Number Theoretic Transform is a finite field analogue of the Discrete Fourier Transform (DFT), operating modulo a prime number. It is especially useful for efficiently performing polynomial multiplication within certain algebraic structures. A common computational strategy for the NTT involves a divide-and-conquer approach, utilizing the Chinese Remainder Theorem (CRT) to break down polynomials into simpler components. Consider the ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(\Phi(X))$, where $\Phi(X)$ is an irreducible cyclotomic polynomial of degree N. This polynomial can be factored into pairwise coprime linear terms as $\Phi(X) = \Pi_{i=0}^{N-1}(X - W_i)$. This factorization leads to an isomorphism given by:

$$\mathbb{Z}_Q[X]/\left(\Phi(X)\right) \cong (\mathbb{Z}_Q[X]/(X - W_0)) \times (\mathbb{Z}_Q[X]/(X - W_1)) \times \cdots \times (\mathbb{Z}_Q[X]/\left(X - W_{N-1}\right))$$

Here, W_i are distinct roots of unity in $\mathbb{Z}_Q$. This isomorphism enables the transformation of polynomial multiplication into point-wise multiplication, which is computationally more efficient. Let $\boldsymbol{f} = \sum_{i=0}^{N-1} f_i X^i$ and $\boldsymbol{g} = \sum_{i=0}^{N-1} g_i X^i$ be polynomials in $\mathcal{R}_Q$. To compute their product modulo $\Phi(X)$, we evaluate each polynomial at the roots W_i, multiply the evaluations point-wise, and then reconstruct the result using an inverse transform.

Let Q be a prime satisfying $Q \equiv 1 \mod N$, there exists a primitive N-th root of unity ω in $\mathbb{Z}_Q$. The forward NTT yield $\hat{\boldsymbol{f}} = \mathsf{NTT}(\boldsymbol{f}) = [\hat{f}_0, \ldots, \hat{f}_{N-1}]$, where $\hat{f}_j = \sum_{i=0}^{N-1} f_i \omega^{ij} \mod Q$. The inverse NTT output $\boldsymbol{f} = \mathsf{INTT}(\hat{\boldsymbol{f}})$, with $f_i = \frac{1}{N} \sum_{j=0}^{N-1} \hat{f}_j \omega^{-ij} \mod Q$.

Cyclic and Negacyclic Convolution. Polynomial multiplication modulo $\Phi(X) = X^N - 1$ and $\Phi(X) = X^N + 1$ corresponds to cyclic and negacyclic convolutions, respectively. While cyclic convolution can be handled using the standard NTT as previously described, negacyclic convolution introduces a sign change in the wrap-around terms, necessitating the use of roots of unity that satisfy $\zeta^{2N} \equiv 1 \mod Q$. For negacyclic convolution, the modulus Q must satisfy $Q \equiv 1 \mod 2N$ to ensure the existence of a primitive $2N$-th root of unity ζ. When applying the Fast Fourier Transform in this context, the forward NTT is defined as $\hat{\boldsymbol{f}} = \mathsf{NTT}(\boldsymbol{f}) = [\hat{f}_0, \ldots, \hat{f}_{N-1}]$, where $\hat{f}_j = \sum_{i=0}^{N-1} f_i \zeta^{(2j+1)i} \mod Q$. The inverse NTT is $\boldsymbol{f} = \mathsf{INTT}(\hat{\boldsymbol{f}})$, with $f_i = \frac{1}{N} \sum_{j=0}^{N-1} \hat{f}_j \zeta^{-(2j+1)i} \mod Q$. This approach reduces the computational complexity of the NTT to $\mathcal{O}(N \log N)$. Consequently, since the point-wise multiplication has a complexity of $\mathcal{O}(N)$, the overall computational complexity of polynomial multiplication is effectively reduced to $\mathcal{O}(N \log N)$.

Four-Step NTT. The four-step algorithm for DFT processing, first introduced by Bailey in [Bai90], decomposes an N-point DFT into smaller, more manageable operations. This technique can be adapted for the NTT to enhance computational efficiency in polynomial multiplication. Consider a polynomial $\boldsymbol{f} = \sum_{i=0}^{N-1} f_i X^i$, where N is factorized as $N = N_1 N_2$. The coefficients are reorganized into a two-dimensional $N_2 \times N_1$ matrix $\mathbf{F}$ defined by $\mathbf{F} = [\boldsymbol{f}_0, \ldots, \boldsymbol{f}_{N_2-1}]^\top$, where each row vector $\mathbf{f}_j$ is $\boldsymbol{f}_j = [f_j, f_{N_2+j}, \ldots, f_{(N_1-1)N_2+j}]$. The four-step NTT algorithm proceeds as follows:

- Step 1: Perform N_2 independent N_1-point NTTs on each $\mathbf{f}_j$ of matrix $\mathbf{F}$, $j \in [0, N_2)$.
- Step 2: Multiply the resulting data by a matrix $\mathbf{W}$ composed of twiddle factors ω^{ij} point-wisely, where ω is the N-th root of unity, $i \in [0, N_1)$, and $j \in [0, N_2)$.
- Step 3: Transpose the resulting $N_2 \times N_1$ matrix to obtain $\mathbf{F}' = [\boldsymbol{f}'_0, \ldots, \boldsymbol{f}'_{N_1-1}]^\top$.
- Step 4: Perform N_1 independent N_2-point NTT on each $\boldsymbol{f}'_i$ of matrix $\mathbf{F}'$, $i \in [0, N_1)$.

This decomposition replaces the computation of a large-scale N-point NTT with a series of smaller N_1-point and N_2-point NTTs, along with element-wise operations and a matrix transposition. By rearranging the elements of the final $N_1 \times N_2$ matrix, one obtains the result of the original N-point NTT of the input polynomial.

NTT with Composite Modulus. NTT can be extended to operate under a composite modulus $Q = \Pi_{i=0}^{K} Q_i$, where each Q_i is a distinct NTT-friendly integer. In such scenarios, the classical method of computing a primitive root of unity is not directly applicable. To overcome this limitation, recent research [HP23, LLL+24] proposes constructing a suitable root of unity by leveraging the roots corresponding to each individual modulus. Specifically, for $Q = Q_1 \cdot Q_2$, let $\zeta_{Q_1,N}$ and $\zeta_{Q_2,N}$ be the $2N$-th primitive root of unity in $\mathbb{Z}_{Q_1}$ and $\mathbb{Z}_{Q_2}$, respectively, we have

$$\zeta_Q = \zeta_{Q_1} \cdot Q_2 \cdot \left(Q_2^{-1} \bmod Q_1\right) + \zeta_{Q_2} \cdot Q_1 \cdot \left(Q_1^{-1} \bmod Q_2\right) \bmod Q \tag{1}$$

With this composite root ζ_Q, the necessary precomputed constants for the NTT under modulus Q can be obtained by calculating the powers.

2.6 GPU

Modern GPUs offer massive parallelism, making them particularly well-suited for accelerating computationally intensive tasks. In this work, we harness GPU parallelism by designing specialized implementations tailored to the GPU architecture, thereby achieving higher performance. A common paradigm in high-performance computing couples a CPU with one or more GPUs to maximize throughput. In this setup, the CPU serves as the host, managing overall program execution, while the GPUs perform parallel computations. The host launches kernels on the GPUs in either synchronous or asynchronous modes. In synchronous mode, the CPU waits until the GPU completes its work, whereas in asynchronous mode, the CPU continues executing other tasks concurrently. Nevertheless, not all operations benefit from GPU acceleration, particularly sequential computations on individual integers in FHEW/TFHE bootstrapping. Consequently, this work adopts asynchronous execution, offloading these sequential components to the CPU so that their latency is effectively masked.

The CUDA programming model provides a direct interface to the GPU's hardware resources and structures them hierarchically. A kernel launch specifies the dimensions of the grid and its constituent blocks, with each grid comprising multiple blocks and each block encompassing up to 1,024 threads. Memory management is crucial for GPU programming. Global memory (GMEM), which resides off-chip, offers ample storage capacity but suffers from high latency. To mitigate this, we use employ on-chip shared memory (SMEM) and registers to store frequently accessed data at the block level and thread level, respectively. We also fuse kernels to maintain intermediate data in on-chip memory, thus improving performance. However, overly aggressive fusion should be avoided, which can decrease the performance. Data access patterns further influence memory interaction latency. In GMEM, coalesced access patterns are key to reducing latency. In SMEM, which is organized into 32 banks, concurrent accesses by multiple threads to the same bank cause bank conflicts, triggering stalls that prevent single-cycle execution. Accordingly, this work arranges memory access patterns carefully to avoid such patterns.

During execution, thread blocks are allocated to streaming multiprocessors (SMs), with 32 threads forming a warp as the basic execution unit for simultaneous processing. The occupancy rate, reflecting the proportion of active warps per SM, indicates resource efficiency. CUDA provides warp-level operations, such as warp shuffle, for synchronized data exchange. Instructions are processed at the warp level, and performance is maximized when all threads execute the same instruction. Divergent branches, where threads in the same warp take different control paths, reduce the number of active threads per cycle, thereby degrading performance. Constant-time implementations not only satisfy security requirements but also improve GPU performance by minimizing thread divergence and ensuring uniform control flow.

3 Hybrid Four-Step NTT

3.1 Design Overview

Previous Methods. Several approaches have been proposed to implement NTT, such as the iterative NTT [ÖEM+22, ÖS23, ÖS24], the merging-level NTT [JKA+21, YSD+24], and the four-step NTT [AMK+23, FWX+23, ÖS23]. In the iterative approach, an N-point NTT is computed level-by-level, where each thread operates on two coefficients and performs a radix-2 butterfly. The merging-level method reduces the overhead of GMEM by aggregating k levels of the NTT into a single stage, thus performing a radix-2^k butterfly on 2^k coefficients. Unlike these techniques, the four-step NTT hierarchically decomposes the transform into N_1-point and N_2-point sub-NTTs, with a matrix multiplication between two phases. Although each of these methods can achieve reasonable performance, several issues arise when they are directly adopted on GPUs:

1. **Memory Overhead.** The iterative NTT demands memory accesses at each level, forcing frequent GMEM interactions especially when the maximum number of threads per block is reached. Merging levels mitigates this cost by relying on SMEM, but still requires $\lceil \log N/k \rceil$ times interactions. Furthermore, the four-step approach accesses data in two dimensions, often inducing memory conflicts that increase IO latency.
2. **Computational Complexity.** Current four-step GPU implementations [FWX+23, ZZF+24] mainly leverage Tensor Cores. Therefore, the sub-NTTs are performed through matrix multiplications that have cubic-time complexity, and demand intricate data slicing to handle 8-bit operations for 64-bit arithmetic. REED [AMK+23] offers linearithmic complexity but partially serializes the second sub-NTT phase to avoid memory conflicts, thus underutilizing parallelism.
3. **Synchronization.** While extensive use of SMEM reduces IO latency [JKA+21], it also incurs thread synchronization overhead before or after each SMEM interaction, which can significantly impact performance.
4. **Thread Alignment.** Many FHEW/TFHE operations are N-parallel. However, radix-2^k butterflies act on groups of 2^k coefficients, making NTT kernels typically launch $N/2^k$ threads. Although kernel fusion can reduce GMEM accesses, the mismatch between N-parallel tasks and 2^k-parallel butterflies complicates thread alignment. This limits the feasibility of kernel fusion. Existing studies [JKA+21, YSD+24] are only able to fuse NTT with constant multiplication.
5. **Scalability.** Optimal configurations for single-task execution often differ from those suited for batched executions. The current designs generally target one specific scenario, rendering them less adaptable to varied workloads with different thread-per-block requirements. Furthermore, some approaches employ fixed parameters, e.g., cuFHE [cuf24], which uses a fixed $N = 2^{10}$ and a Goldilocks prime modulus, making them challenging to extend to alternative parameter settings.
6. **Inconsistent Workload.** Many studies implement non-constant time modular reduction. For instance, in cuFHE [cuf24], a conditional check is used to determine whether a number exceeds Q, followed by a conditional bitmask operation for reduction. This approach results in inconsistent workloads across threads, leading to thread divergence and an increase in execution cycles.

Our Methods. For efficient FHEW/TFHE computations on GPUs, an NTT design that is fully parallelized, memory-efficient, easily fused, and flexible enough to handle both single and batched executions is more demanded. We propose a hybrid four-step NTT framework that addresses all these requirements. First, to overcome Limitations 1 and 2, we generalize the four-step framework to support both cyclic and negacyclic convolutions, preserving a compact structure and linearithmic complexity. Motivated by a more memory-efficient external-product technique [LLL+24], our design remains

Table 1: Comparison of different NTT implementations. The parallelism shows each design's maximum achievable parallelism. For GMEM access, only accesses to the input polynomial are counted.

	[ÖEM+22]	[SYL+23]	[YSD+24], [JKA+21]	[FWX+23]	This work
Type	Radix-2	Radix-2	Radix-8	Tensor core	Warp shuffle/ Radix-2
Parallelism	$N/2$	$N/2$	$N/8$	N	N
Complexity	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N_1 + N \log N_2)$	$\mathcal{O}(N \log N_1 + N \log N_2)$	$\mathcal{O}(N(N_1 + N_2) + N_1 N_2)$	$\mathcal{O}(N \log N_1 + N \log N_2)$
GMEM access	$2(\log N - 5)N$	$4N$	$4N$	$10N$	$2N$
Synchronization	$\log N$	$\lceil \log N_1/3 \rceil + \log N_2 - 5$	$\lceil \log N/3 \rceil$	2^*	2

*Estimated.

compatible with composite moduli used in such approaches. Second, we propose a warp-based GPU implementation to alleviate Limitations 3, 4 and 5. Through a novel warp-shuffle strategy, we enable N-parallel NTT computations that align with other GPU kernels, and minimize synchronizations. This design facilitates broader kernel fusion (e.g., merging the entire accumulation in blind rotation) and accommodates flexible thread block configurations, thus optimizing performance across various scenarios. We adopt a constant-time strategy to ensure consistent workloads across threads, thereby avoiding thread divergence that can degrade performance, solving Limitation 6. Third, we introduce memory-centric optimizations to tackle Limitation 1 without hindering the parallelizability of sub-NTTs mentioned in Limitation 2, ensuring fully parallelized sub-NTT execution. By incorporating these optimizations, our approach achieves near-theoretical maximum SM occupancy, leading to superior hardware utilization. A comparative summary of these designs is presented in Table 1.

3.2 Proposed Hybrid Four-Step NTT with Linearithmic Complexity

Below we introduce our hybrid four-step NTT design. We begin by describing the standard decomposition approach, which is typically realized via matrix multiplication. Next, we adapt this approach to guarantee linearithmic complexity and to support both cyclic and negacyclic convolutions, a feature that proves advantageous for diverse parameter configurations. Finally, we discuss how to incorporate composite moduli into our design, thereby extending its applicability to a broader range of use cases.

Let ζ_N denote the $2N$-th primitive root of unity in $\mathbb{Z}_Q$, and let ζ_{N_1} and ζ_{N_2} denote the $2N_1$-th and $2N_2$-th primitive roots of unity, respectively. We have $\zeta_{N_1} = \zeta_N^{N_2} \bmod Q$ and $\zeta_{N_2} = \zeta_N^{N_1} \bmod Q$. For indices $i, y \in [0, N_1)$ and $j, z \in [0, N_2)$, the forward and inverse NTT can be formulated as follows:

$$
\begin{aligned}
\hat{f}_{y+N_1 z} &= \sum_{j=0}^{N_2-1} \sum_{i=0}^{N_1-1} f_{iN_2+j} \cdot \zeta_N^{2(iN_2+j)(y+N_1 z)+(iN_2+j)} \\
&= \sum_{j=0}^{N_2-1} \left(\zeta_N^{2jy+j} \cdot \left(\sum_{i=0}^{N_1-1} f_{iN_2+j} \cdot \zeta_{N_1}^{2iy+i} \right) \right) \cdot \zeta_{N_2}^{2jz} \mod Q \\
f_{iN_2+j} &= \frac{1}{N_1 N_2} \cdot \sum_{y=0}^{N_1-1} \sum_{z=0}^{N_2-1} \hat{f}_{y+N_1 z} \cdot \zeta_N^{-2(iN_2+j)(y+N_1 z)-(iN_2+j)} \\
&= \frac{1}{N_1 N_2} \cdot \sum_{y=0}^{N_1-1} \zeta_{N_1}^{-2iy-i} \cdot \left(\zeta_N^{-2jy-j} \cdot \left(\sum_{z=0}^{N_2-1} \hat{f}_{y+N_1 z} \cdot \zeta_{N_2}^{-2jz} \right) \right) \mod Q
\end{aligned}
\tag{2}
$$

In this formulation, the forward transform can be achieved through multiplication with three matrices: $\mathbf{W}_1 := \{\zeta_{N_1}^{2iy+i}\}_{N_1\times N_1}$, $\mathbf{W}_2 := \{\zeta_N^{2jy+j}\}_{N_1\times N_2}$, and $\mathbf{W}_3 := \{\zeta_{N_2}^{2jz}\}_{N_2\times N_2}$. This method incurs computational complexities of $\mathcal{O}(N_1^2)$, $\mathcal{O}(N_1N_2)$, and $\mathcal{O}(N_2^2)$ in each phase, respectively, with a similar approach for the inverse transform. This strategy is well-suited for computational units optimized for matrix multiplication, such as Tensor Cores. Conversely, when utilizing traditional hardware, methods that offer lower computational complexity are preferable for NTT computation. By modifying Equation (2), we can compute the N_1-point and N_2-point NTTs with linearithmic complexity. Specifically, we derive two variants that integrate different types of convolution:

1) Negacyclic Convolution. We modify Step 3 by multiplying with the matrix $\mathbf{W}_2' = \{\zeta_N^{(2y+1-N_1)j}\}_{N_1\times N_2}$, so that both sub-NTT phases can be adjust to perform negacyclic convolutions. Utilizing Fast Fourier Transform techniques, we can achieve computational complexity of $\mathcal{O}(N_1 \log N_1)$ and $\mathcal{O}(N_2 \log N_2)$, respectively. This method is especially beneficial when $N_1 = N_2$, as it allows for identical NTT submodules and precomputed constants, requiring only a single precomputed table in SMEM. Specifically, this can be formulated as:

$$\begin{aligned}
\hat{f}_{y+N_1z} &= \sum_{j=0}^{N_2-1}\left(\zeta_N^{2jy+j}\cdot\left(\sum_{i=0}^{N_1-1} f_{iN_2+j}\cdot\zeta_{N_1}^{2iy+i}\right)\right)\cdot\zeta_{N_2}^{2jz+j}\cdot\zeta_{N_2}^{-j} \\
&= \sum_{j=0}^{N_2-1}\left(\zeta_N^{(2y+1-N_1)j}\cdot\left(\sum_{i=0}^{N_1-1} f_{iN_2+j}\cdot\zeta_{N_1}^{2iy+i}\right)\right)\cdot\zeta_{N_2}^{2jz+j} \bmod Q \\
f_{iN_2+j} &= \frac{1}{N_1N_2}\cdot\sum_{y=0}^{N_1-1}\zeta_{N_1}^{-2iy-i}\cdot\left(\zeta_N^{-2jy-j}\cdot\left(\sum_{z=0}^{N_2-1}\hat{f}_{y+N_1z}\cdot\zeta_{N_2}^{-2jz-j}\cdot\zeta_{N_2}^{j}\right)\right) \\
&= \frac{1}{N_1N_2}\cdot\sum_{y=0}^{N_1-1}\zeta_{N_1}^{-2iy-i}\cdot\left(\zeta_N^{(-2y-1+N_1)j}\cdot\left(\sum_{z=0}^{N_2-1}\hat{f}_{y+N_1z}\cdot\zeta_{N_2}^{-2jz-j}\right)\right) \bmod Q
\end{aligned} \tag{3}$$

2) Hybrid Convolution. Different from negacyclic convolution, cyclic convolution halves the twiddle factors accessed during computation, reducing memory requirements. Let ω_{N_2} be the N_2-th root of unity, thus $\omega_{N_2} = \zeta_{N_2}^2$, we derive:

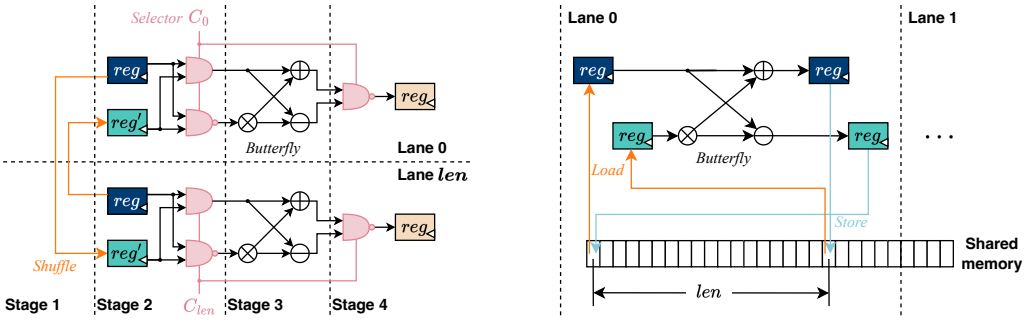
$$\begin{aligned}
\hat{f}_{y+N_1z} &= \sum_{j=0}^{N_2-1}\left(\zeta_N^{2jy+j}\cdot\left(\sum_{i=0}^{N_1-1} f_{iN_2+j}\zeta_{N_1}^{2iy+i}\right)\right)\omega_{N_2}^{jz} \mod Q \\
f_{iN_2+j} &= \frac{1}{N_1N_2}\cdot\sum_{y=0}^{N_1-1}\zeta_{N_1}^{-2iy-i}\cdot\left(\zeta_N^{-2jy-j}\cdot\left(\sum_{z=0}^{N_2-1}\hat{f}_{y+N_1z}\omega_{N_2}^{-jz}\right)\right) \mod Q
\end{aligned} \tag{4}$$

This makes it more suitable for scenarios where $N_1 \neq N_2$. It's equally applicable when $N_1 = N_2$, as $\omega_{N_2}^i$ can be derived from the twiddle factors generated by ζ_{N_1}.

Adaptation for Composite Moduli. Considering recent progress in computing external product under composite moduli [LLL+24], we adapt our four-step NTT design to handle this setting. While the computational logic remains largely the same, adjustments are needed in constructing the roots of unity. Given $Q = Q_1Q_2$, where primes Q_1 and Q_2 are NTT-friendly, let $\zeta_{Q_1,N}$ and $\zeta_{Q_2,N}$ be the $2N$-th primitive roots of unity in $\mathbb{Z}_{Q_1}$ and $\mathbb{Z}_{Q_2}$, respectively. Using Equation (1), we construct the $2N$-th primitive root $\zeta_{Q,N}$. By decomposing $N = N_1N_2$, we compute the corresponding roots as $\zeta_{Q,N_1} = \zeta_{Q,N}^{N_2} \bmod Q$ and $\zeta_{Q,N_2} = \zeta_{Q,N}^{N_1} \bmod Q$. These roots facilitate efficient computation of the smaller N_1-point and N_2-point NTTs under the composite modulus Q.

3.3 Implementation and Optimizations

In this section, we present the implementation details of our proposed hybrid four-step NTT design. First, we develop specialized strategies for sub-NTT modules of different sizes, ensuring high utilization of GPU resources. Second, to address the substantial memory access overhead and potential conflicts, we refine data storage and access patterns. This enables a compact single-kernel NTT implementation with minimized global memory usage and synchronization overhead. Finally, we demonstrate the scalability of our design by allowing flexible configurations of thread blocks, facilitating performance optimization across diverse devices and parameter sets.



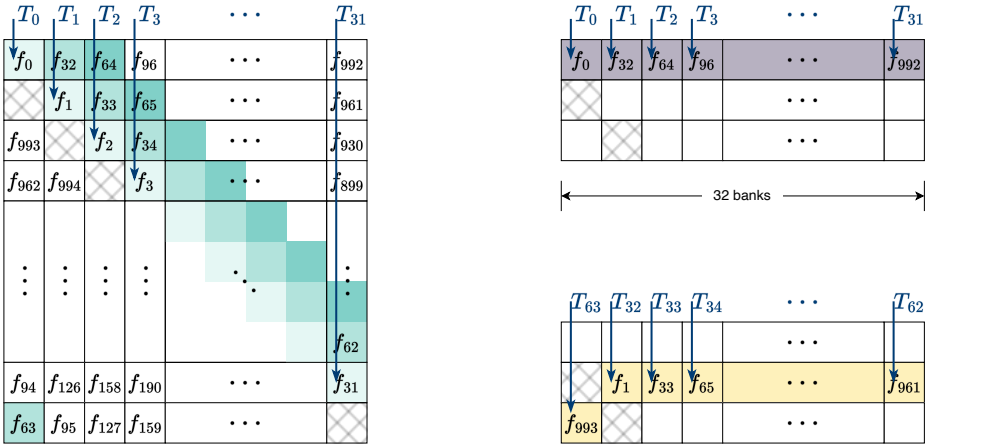
(a) 32-point NTT using warp shuffle. **(b)** 64-point NTT using radix-2 approach.

Figure 1: Implementation of 32-point and 64-point NTT modules.

Implementation Details. In FHEW and TFHE, the ring dimension N typically takes values $N \in \{2^{10}, 2^{11}, 2^{12}\}$. We decompose N into combinations of $N_1, N_2 \in \{32, 64\}$, enabling implementation of a 32-point and a 64-point fundamental NTT module. Each module utilizes a single warp to process input of size 32 or 64, respectively, allowing efficient batched processing of multiple NTTs. Specifically, we devise the following approaches for sub-NTT implementation, which are then composed for larger dimensions.

1) 32-Point NTT. The computational workflow of our 32-point NTT is illustrated in Figure 1a. Since traditional radix-2^k NTT implementation only utilize $N/2^k$ threads, thus for 32-point NTT it makes threads within a warp not fully utilized. Thus, we devise a warp-shuffle based implementation, with a constant-time conditional multiplexer (CMUX) for data exchanges across threads. In our implementation consists of four stages. Initially, each thread in a warp holds one coefficient of the input polynomial in its register, denoted as a_i for the i-th thread. First, in the data shuffling stage, we calculate the distance between paired coefficients, denoted as len. Each thread then uses the `__shfl_xor_sync` instruction to retrieve the coefficient a'_i from a neighboring thread at a distance of len, storing it in a new register. Second is input selection, here the left input is computed as $a_l = (1-C)\cdot(a_i - a'_i) + a'_i$, and the right input as $a_r = C\cdot(a_i - a'_i) + a'_i$, where the selector C is determined by the lane index and len. Third, we perform the butterfly computation using the selected inputs. Finally is the output selection, we update a_i for the next level through $a_i \leftarrow (1-C)\cdot(a_l - a_r) + a_r$. This ensures consistent workload across threads, preventing divergence and maintaining high performance.

2) 64-Point NTT. For the 64-point NTT, we adopt the radix-2 implementation, as depicted in Figure 1b. Unlike the 32-point NTT, this design is more straightforward and fully utilizes all 32 threads in a warp. The input polynomial is stored in SMEM to expedite data access during computation. In each of the six NTT levels, every thread loads two coefficients from SMEM, performs the butterfly operation, and writes the results back to SMEM. This approach effectively balances computational workload and resource



(a) Visualization of data storage in SMEM that achieves implicit matrix transposition.

(b) Illustration of the conflict-free data access pattern during sub-NTT computations.

Figure 2: Memory layout and access patterns in our optimized NTT implementation.

utilization, leading to potential advantage in operation fusing.

Memory-Tailored Optimizations. Although the size of the polynomials involved allows them to fit within SMEM, the four-step NTT approach presents challenges in data access patterns. The initial N_1-point NTT operates on coefficients spaced at intervals of N_2, leading to strided GMEM accesses if the data is not rearranged. However, if rearranging the polynomial using SMEM, it not only introduce 32-way bank conflicts to either reading or writing, but also makes the entire NTT computation requires three matrix transpositions. To address these issues, we explore access pattern-driven optimizations to simplify and enhance memory access efficiency.

1) Implicit Matrix Transposition. Since GMEM access is the most time-consuming operation, we ensure coalesced reads by loading the polynomial data sequentially. Storing the data in SMEM is essential, so we devise a storage method that performs an implicit matrix transposition. Specifically, we manipulate the data layout in SMEM by writing in row-major order but distributing the data across different rows and memory banks, as shown in Figure 2a. This approach achieves an implicit transposition while maintaining optimal access patterns for both GMEM reads and SMEM writes, allowing all 32 threads in a warp to access data in a single cycle. Additionally, this design eliminates the explicit transpose step typically required in the four-step NTT approach.

2) Conflict-Free Data Access. The modified memory layout effectively introduces zero-padding of one bank unit every 32 data elements, which facilitates conflict-free access during sub-NTT computations. Figure 2b illustrates the data layout and access pattern for a 32-point NTT. In this scheme, threads in the first warp process coefficients $[f_0, f_{32}, \ldots, f_{992}]$ to compute the N_1-point NTT, with all 32 coefficients residing in the first row of SMEM and accessible in a single cycle. Threads in the second warp handle coefficients $[f_1, f_{33}, \ldots, f_{993}]$, which are also stored in different banks. This design is particularly advantageous for the 64-point NTT, where each level requires two interactions with SMEM. By ensuring efficient data access, we can retrieve the necessary 64 coefficients in just two cycles.

3) Inter-Kernel Fusing. A straightforward implementation of the full N-point NTT might involve three separate kernels: the N_1-point NTT, the matrix multiplication, and the N_2-point NTT. However, since the matrix multiplication consists of simple point-wise modular multiplications, we can integrate it into the N_1-point NTT kernel. Upon

completing the N_1-point NTT, we retain the results in registers and perform the modular multiplication directly. Furthermore, leveraging the implicit matrix transposition achieved through SMEM, we can fuse the N_1-point and N_2-point NTTs into a single kernel. This integration significantly reduces memory accesses, as temporary data remains in registers or SMEM, which offer faster access speeds. Additionally, this fused kernel can be further integrated with other operations in blind rotation and bootstrapping, enhancing overall performance.

Scalable Block Fine-Tuning. For single NTT computation, achieving the maximum parallelism is optimal. However, when batch processing multiple computations, the thread block instantiation that offer best performance varies with device specifications and polynomial size. Fine-tuning the thread block dimension is crucial for peak performance. Our warp-based approach scales efficiently to different parallelism levels. By using one warp per sub-NTT computation, we can simultaneously compute with $D/32$ warps when using D threads ($D \leq N$) to instantiate a thread block. The computation completes through $\lceil N_2/(D/32) \rceil$ loops for the first stage and $\lceil N_1/(D/32) \rceil$ for the third stage.

3.4 Discussions

Compatibility with FFT. Recent study [XLK+25] has observed that using FFT within the FHEW/TFHE bootstrapping procedure can sometimes be more efficient than NTT. Notably, our proposed four-step NTT framework remains fully compatible with FFT due to the conceptual similarity between the two transforms, since the four-step framework was originally devised for FFT [Bai90], and NTT is a variant of FFT over a finite field. Consequently, the strategies we employed in designing and optimizing our NTT can be adapted for FFT by adjusting data types and numerical operations to accommodate floating-point arithmetic instead of modular arithmetic. Furthermore, modern GPUs excel at processing floating-point operations, largely because of their specialized hardware support. Therefore, our GPU-oriented optimizations, such as minimizing GMEM interaction, reducing synchronization overhead, and maximizing resource utilization, could be leveraged to further enhance the performance of FFT-based operations.

Generalization and Applicability. Unlike word-wise FHE schemes, which often involve several homomorphic operations at the ciphertext level, bit-wise FHE schemes rely primarily on bootstrapping for homomorphic evaluation. Consequently, our proposed NTT implementation will be most relevant for bit-wise FHE in the bootstrapping phase. Nevertheless, its design remains broadly applicable to a variety of scenarios, offering flexibility in parameter settings. In particular, our four-step NTT approach is easily extended to word-wise FHE schemes that require NTT computations in nearly all homomorphic operations. By enabling N in the range of 2^{13} to 2^{17}, we can offer larger NTT computations in these FHE schemes. From a theoretical perspective, choosing N_1 and N_2 close to $\sqrt{N}$ minimizes computational complexity in the four-step framework. However, practical considerations, such as tighter integration of sub-NTT modules with subsequent computations, may lead to alternative choices of N_1 and N_2. For instance, one may fix N_2 and design that sub-NTT to be well-fused with subsequent operations, while adjusting N_1 to accommodate different parameter sets. This strategy can reduce memory access overhead and achieve notable performance gains under specific constraints, illustrating the adaptability of our design across diverse FHE workloads.

Algorithm 2 Optimized AP blind rotation using modulus raising

Input: LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, bootstrapping key $\mathbf{bsk}$.

Output: LWE ciphertext $\mathbf{acc} \in \mathsf{RLWE}_{\mathbf{s}'}^{N,Q}(X^{b-\sum_{i=0}^{n-1} a_i s_i})$

Set $\mathbf{acc}_0 = (\mathbf{0}, X^b) \in \mathcal{R}_Q^2$
for $i \in [1, n]$ **do**
  for $j \in [0, \log_{B_r} q)$ **do**
    $a_{i,j} \leftarrow \lfloor a_i / B_r \rceil \bmod B_r$
    $\widehat{\mathbf{acc}}_i \leftarrow \mathsf{4StepCompNTT}(\mathbf{acc}_{i-1})$
    $\widehat{\mathbf{tmp}}_i \leftarrow \widehat{\mathbf{acc}}_i \odot \mathbf{bsk}_{i,j,a_{i,j}} \in \mathcal{R}_{PQ}^2$
    $\mathbf{tmp}_i \leftarrow \mathsf{4StepCompINTT}(\widehat{\mathbf{tmp}}_i)$
    $\mathbf{acc}_i \leftarrow \lfloor \mathbf{tmp}_i / P \rceil \in \mathcal{R}_Q^2$
return $\mathbf{acc}_n$

Algorithm 3 Optimized GINX blind rotation using modulus raising

Input: LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, bootstrapping key $\mathbf{bsk}$.

Output: LWE ciphertext $\mathbf{acc} \in \mathsf{RLWE}_{\mathbf{s}'}^{N,Q}(X^{b-\sum_{i=0}^{n-1} a_i s_i})$

Set $\mathbf{acc}_0 = (\mathbf{0}, X^b) \in \mathcal{R}_Q^2$
for $i \in [1, n]$ **do**
  $\widehat{\mathbf{acc}}_i \leftarrow \mathsf{4StepCompNTT}(\mathbf{acc}_i)$
  $\widehat{\mathbf{tmp}}_i \leftarrow ((X^{a_i} - 1) \odot \widehat{\mathbf{acc}}_i) \odot \mathbf{bsk}_{i,0} + ((X^{-a_i} - 1) \odot \widehat{\mathbf{acc}}_i) \odot \mathbf{bsk}_{i,1} \in \mathcal{R}_{PQ}^2$
  $\mathbf{tmp}_i \leftarrow \mathsf{4StepCompINTT}(\widehat{\mathbf{tmp}}_i)$
  $\mathbf{acc}_i \leftarrow \mathbf{acc}_i + \lfloor \mathbf{tmp}_i / P \rceil \in \mathcal{R}_Q^2$
return $\mathbf{acc}_n$

4 Optimizations in Bootstrapping

4.1 Blind Rotation

There are two primary strategies for performing the external product in both AP- and GINX-type blind rotations, i.e., the gadget decomposition approach [DM15, CGGI16] and the modulus raising approach [LLL+24]. The former requires decomposing ciphertexts into multiple smaller slices, which results in significant data expansion and incurs considerable memory overhead. In addition, it necessitates numerous NTT calls because each slice must be multiplied by the bootstrapping key, thereby introducing a substantial performance bottleneck on GPUs. By contrast, the modulus raising approach, which raises the modulus from Q to PQ for the external product, circumvents ciphertext decomposition and reduces the number of NTT calls. Nonetheless, its noise control is weaker than that of the gadget decomposition method.

Our Optimizations in Blind Rotation. Taking into account the advantages and disadvantages of both approaches, VeloFHE includes both techniques. In addition, we propose several optimizations to further improve performance and noise management.

1) Memory Optimization for Accumulation. Our compact NTT design not only supports fusing NTT into a single kernel but also enables integrating the entire accumulation process into one kernel invocation. This strategy is applicable to both the gadget decomposition and modulus raising approaches. By consolidating the accumulator updates into this single kernel, we avoid substantial GMEM interactions. Therefore, the primary memory requests are directed to SMEM and the register file, thereby reducing overall IO latency and alleviating the memory bottleneck.

In addition, we modify the bootstrapping keys to be generated in the Montgomery domain, allowing us to employ Montgomery modular multiplication [Mon85]. By devising a finely optimized PTX assembly implementation, we reduce the instruction count compared

to the Barrett reduction approach [Bar86].

2) Reduce Noise Growth in Modulus Raising Approach. Unlike the method proposed in [LLL+24], which performs accumulation in the NTT domain over $\mathcal{R}_{PQ}$, we propose carrying out it in the normal domain over $\mathcal{R}_Q$. In this variant, only the element-wise multiplication takes place in $\mathcal{R}_{PQ}$, after which the result is downscaled by P before updating **acc**. The details of the computations are shown in Algorithms 2 and 3, respectively. This design offers two primary benefits. First, the rounding operation is applied only to $\widehat{\mathbf{tmp}}$ rather than the entire accumulator, thereby mitigating noise introduced during this computation. Second, maintaining the accumulator in $\mathcal{R}_Q$ preserves compatibility, particularly when handling unsigned integers, thus eliminating additional processing required for converting negative numbers between $\mathcal{R}_{PQ}$ and $\mathcal{R}_Q$.

4.2 Key-Switching

Following sample extraction, which produces an LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}'}^{N,Q}(m)$, the key-switching operation converts the encryption key from $\mathbf{s}'$ to $\mathbf{s}$. Since the gadget decomposition base for $\mathbf{a}$ is fixed, the key-switching key **ksk** can be precomputed. Consequently, this process consists of decomposing the ciphertext, loading the key elements based on the decomposition digits, and accumulating the results onto $(\mathbf{0}, b)$ to produce the new ciphertext $\mathbf{ct}' \in \mathsf{LWE}_{\mathbf{s}}^{n,Q}(m)$. Although the procedure is conceptually straightforward, GPU implementation introduces multiple challenges, and we propose the following solutions. An overview of our implementation is presented in Figure 3.

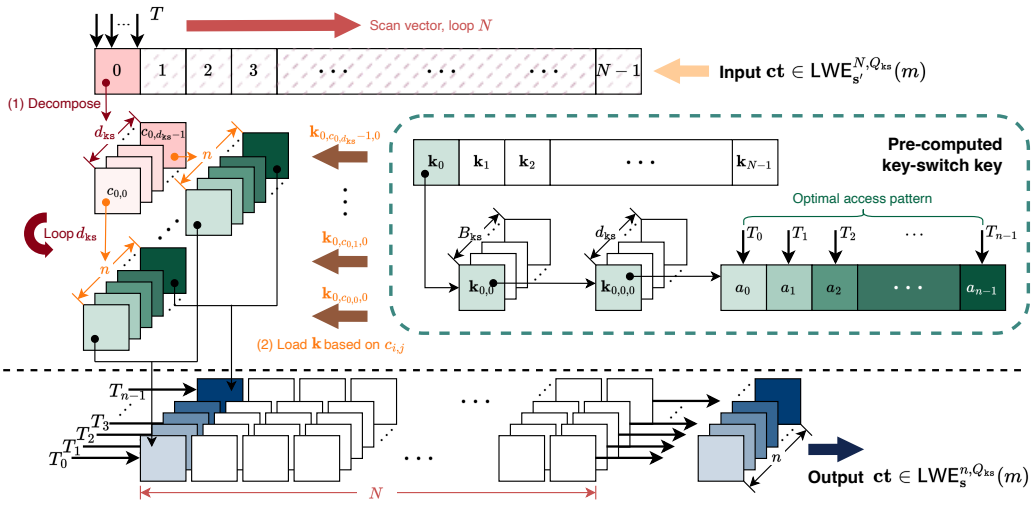


Figure 3: Overview of our implementation of key-switching operation on GPU. Here, T_i $(i \in [0, n))$ denotes the i-th thread, and T indicates that the process is applied across all threads indexed from 0 to N.

1) Dimension-Aware Adjustment for Input-Output Mismatch. A major challenge arises from the difference between the input vector size N and the output size n. When assigning these operations to GPU threads within a single kernel, such mismatches can lead to either redundant thread computations or diminished overall parallelism. Furthermore, uncoalesced memory accesses to **ksk** can significantly degrade performance if the data layout is not carefully managed. To address these issues, we propose a dimension-aware adjustment to the baseline computational flow and align the parallelization with n. This strategy enables each thread to load key elements from global memory in a coalesced pattern and prevents redundant thread-level computations.

2) Asynchronous Execution to Mitigate Workload Imbalance. Another bottleneck arises because a single thread is responsible for processing the integer b, a scalar quantity, potentially causing other threads to stall until that operation concludes. This imbalance underutilizes GPU resources and reduces throughput. To alleviate this, we delegate the computation involving b to the CPU, which is better suited for such sequential tasks. By executing the GPU and CPU components asynchronously, the GPU continues uninterrupted with the $\mathbf{a}$-based computations, while the CPU handles b simultaneously. This overlapping execution effectively hides the latency of the scalar processing on the CPU and improves GPU occupancy.

4.3 Functional Bootstrapping

For the design of bootstrapping procedure, we address two primary considerations. First, at the operation level, we require a unified framework that supports both the AP and GINX variants, using either gadget decomposition or modulus raising. Second, since many real-world applications demand processing multiple LWE ciphertexts simultaneously, efficient batching mechanisms are essential. We incorporate a series of optimizations to meet these requirements.

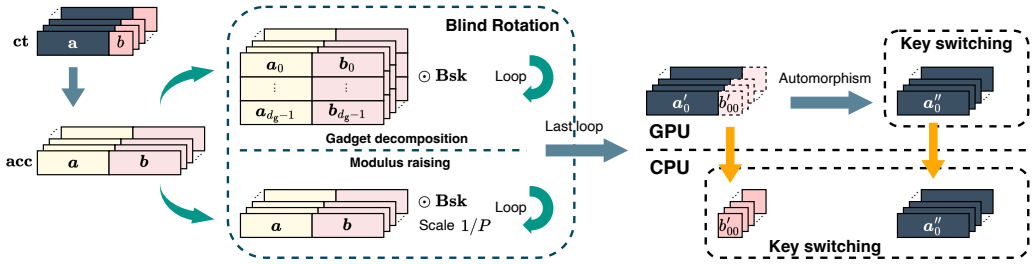


Figure 4: Computational flow of the bootstrapping module in our VeloFHE implementation.

Framework Design. Our bootstrapping workflow consists of blind rotation, sample extraction, modulus switching to a temporary modulus $Q_{\mathtt{ks}}$, key switching, and a final switch back to modulus q, as illustrated in Figure 4. To minimize both memory accesses and computational overhead, we employ the following strategies:

- We consolidate the blind rotation into a single kernel. Additionally, the element-wise type modulus switching operations are fused into the sample extraction and key switching steps.
- In the accumulator $\mathbf{acc} = (\boldsymbol{a}, \boldsymbol{b})$, where $\boldsymbol{b} = \sum_{i=0}^{N-1} b_i X^i$ is a polynomial, only the constant term b_0 is needed for subsequent computations. Consequently, the computations associated with b_1 to b_{N-1} can be omitted. We therefore compute only b_0 in the final stage of the INTT, thereby reducing computational load.

Batching and Workload Partition. Let $\mathbf{ct}'_i = (\mathbf{a}'_i, b'_i)$ represent the ciphertexts obtained after blind rotation, where $i \in [0, \beta)$ and β is the batch size. Since we offload the accumulation of the integers b'_i during key switching to the CPU, we encounter the issue of copying multiple b'_i values to the CPU. Due to their non-contiguous locations in memory, this transfer can be inefficient. To address this challenge, we implement a memory pooling mechanism in GMEM that separates the storage of $\mathbf{a}'_i$ and b'_i. This design ensures that all b'_i values are stored contiguously, allowing a single copy instruction to efficiently transfer them to the CPU. Meanwhile, we asynchronously launch a stream on the GPU to process $\mathbf{a}'_i$. Moreover, since the CPU requires information about $\mathbf{a}'_i$ to load the corresponding key-switching keys during the accumulation over b'_i, we transfer the automorphism of

$\mathbf{a}'_i$, which is the primary computation in sample extraction, to the CPU. As a result, key switching on $\mathbf{a}'_i$ and b'_i proceeds independently on the GPU and CPU, respectively, effectively hiding the latency associated with the CPU-based scalar operations.

5 Experimental Results

We compile the C/C++ code using g++ 12.2.0 and the GPU implementations with CUDA 11.8 on an Arch Linux system with kernel 5.15. We present GPU performance results on a NVIDIA Tesla A100 80G PCIe and a NVIDIA GeForce RTX 4090. CPU baseline is measured on an Intel(R) XEON W7-2495X CPU with 24 cores. The latency of host-device data transfer is also included. Table 2 shows parameter sets used in our evaluation of bootstrapping, covering both gadget decomposition and modulus raising methods. The GD-I/II correspond to the standard `STD128` and `STD128Q_3` parameter sets in OpenFHE [ope24], and the MR-I/II are derived from [LLL+24], for fair comparison with CPU implementations. All parameter sets achieve 128-bit security [APS15]. For comparison with related work [XLK+25], we adopt the same parameter configurations used in their study to ensure a fair comparison.

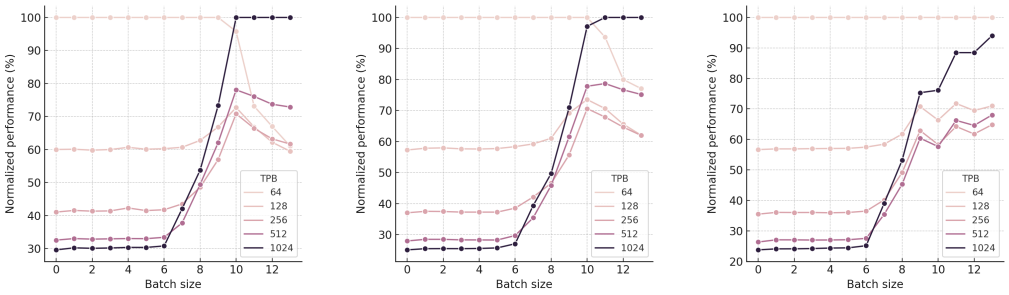
Table 2: Parameter sets for both FHEW and TFHE. Here, "GD" denotes the implementation using gadget decomposition, and "MR" indicates using modulus raising approach.

Parameters	n	q	N	$\log Q$	$B_{\mathtt{g}}$	$B_{\mathtt{ks}}$	$Q_{\mathtt{ks}}$
GD-I	503	2^{10}	2^{10}	27	2^8	2^5	2^{14}
GD-II	600	2^{11}	2^{11}	50	2^{25}	2^5	2^{15}
MR-I	2^9	2^9	2^{10}	16 ($\log P = 20$)	-	2^9	2^{14}
MR-II	2^{10}	2^{10}	2^{11}	22 ($\log P = 28$)	-	2^5	2^{21}

5.1 Performance of Building Blocks

In the following, we present performance results for our building blocks and demonstrate the effectiveness of the proposed optimizations. We begin by fine-tuning thread block instantiation to determine the optimal configurations, which both ensures high performance and illustrates the scalability of our implementation. Next, we discuss the performance of our NTT implementation and compare it with related work. Finally, we highlight the performance improvements gained from our memory-tailored optimizations.

Fine-Tuning for Batch Processing. The dimensions of thread blocks significantly impact performance and vary across different GPU architectures. To optimize our hybrid four-step NTT implementation, we examined its sensitivity to threads per block (TPB). Figure 5 presents the normalized execution times of our NTT implementation for various TPB settings across different batch sizes. The results reveal consistent trends. Setting the TPB to the maximum of 1024 yields the best performance for smaller batch sizes, of which the execution time is only 25% to 30% of the maximum execution time. As batch size increases, the optimal TPB decreases, shifting from 512 to 256 or 128. This reflects the increased efficiency of smaller TPB values in improving SM occupancy, leading to performance gains. The turning points occur around batch sizes of 2^{10} and 2^9 for the three N values, with optimal results achieved at TPB values of 512, 256, or 128. This occurs when GPU hardware resources are fully utilized, favoring configurations that enhance occupancy. This shows that our implementation scales well to varying TPB configurations, it allows for optimal performance across different parameter sets and batch sizes.



(a) $N = 2^{10}$ **(b)** $N = 2^{11}$ **(c)** $N = 2^{12}$

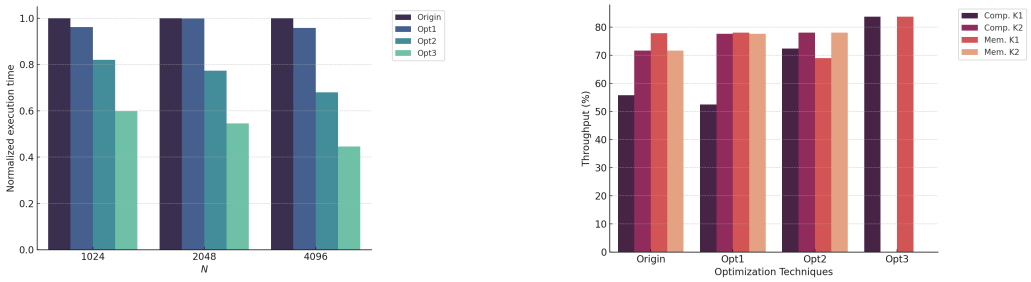
Figure 5: Execution time analysis of our hybrid four-step NTT implementation in VeloFHE across varying batch sizes. The batch size is represented as powers of two. The y-axis shows the execution time normalized to the maximum within each batch size. The TPB denote number of threads per block.

NTT Performance and Comparisons. We compare our implementation to existing works under the same 64-bit arithmetic setting, which is a common choice in NTT implementations for FHE schemes. By applying our fine-tuning procedure, we determine the optimal threads per operation (TPO) for our NTT implementation. Table 3 presents the amortized execution times of our implementation and comparisons with [cuf24, ÖS23], both of which provide open-source GPU-based NTT implementations. For fair comparisons, we align the batch size β and execute all codes on the same A100 GPU. The NTT implementation in cuFHE [cuf24] is limited to $N = 1024$ applies TPO of 128, employing a Goldilocks prime modulus to enable fast modular reduction via bit shifting, and batching is achieved by launching multiple CUDA streams. In contrast, our implementation supports more general sizes and moduli, thus allowing flexible parameter configurations. Under our optimal TPO, our approach achieves speedups of 39.1% and 43.5% for NTT and INTT, respectively. Moreover, the work in [ÖS23] provides a four-step NTT over $\mathbb{Z}_Q[X]/(X^N - 1)$, as used in the HEonGPU library [ÖS24]. Despite both implementations employing a similar four-step framework, our implementation strategy and more fine-grained optimizations yield performance improvements of 35.7% and 38.3% for NTT and INTT, respectively. These results underscore the effectiveness of our design and optimizations.

Table 3: Execution time of our hybrid four-step NTT implementation utilizing 64-bit arithmetic, with comparisons to [YSD+24, ÖS23], on an A100 GPU. β represents the batch size, and TPO denotes the number of threads per operation. Speedup is calculated relative to implementations with identical parameters.

	Execution time (μs)									Improvement	
	VeloFHE							[cuf24]	[ÖS23]	[cuf24]	[ÖS23]
N	2^{10}	2^{10}	2^{10}	2^{11}	2^{11}	2^{12}	2^{12}	2^{10}	2^{12}		
β	1	1024	8192	1	1024	1	1024	1	1024		
TPO	1024	256	128	1024	256	1024	512	128	2048		
NTT	**12.0**	0.061	0.049	15.6	0.090	17.9	**0.117**	19.7	0.182	**39.1%**	**35.7%**
INTT	**12.1**	0.059	0.049	16.0	0.090	19.8	**0.129**	21.4	0.209	**43.5%**	**38.3%**

Effectiveness of Optimizations. Figure 6 illustrates the impact of our NTT optimizations through normalized execution times, compute throughput, and memory throughput across three stages. The original implementation utilizes two kernels for N_1-point NTT fused



(a) Normalized execution time.

(b) Compute and memory throughput.

Figure 6: Performance breakdown of NTT in VeloFHE. (a) Normalized execution time for each N after successive application of optimization techniques. (b) Compute (Comp.) and memory (Mem.) throughput for $N = 1024$ following each optimization stage.

Table 4: Latency and throughput results of gate bootstrapping of FHEW and TFHE on CPU and our VeloFHE implementation on GPU. CPU execution times for FHEW and TFHE GD-I/II are derived from OpenFHE [ope24] using 48 threads. TFHE timings under MR-I/II are sourced from [LLL+24].

Parameter sets	Scheme	CPU	A100 GPU		RTX 4090 GPU		Speedup
			Time	Throughput	Time	Throughput	
		(ms)	(ms)	(OPS)	(ms)	(OPS)	
GD-I	FHEW	52.06	0.22	4,545	0.14	7,007	**371.86**×
	TFHE	35.14	0.14	7,190	0.09	11,378	**390.44**×
GD-II	FHEW	121.12	0.56	1,781	0.41	2,414	**295.41**×
	TFHE	86.59	0.37	2,686	0.31	3,249	**279.32**×
MR-I	FHEW	-	0.28	3,549	0.16	6,439	-
	TFHE	34.38	0.17	5,969	0.10	9,871	**343.80**×
MR-II	FHEW	-	0.97	1,030	0.78	1,279	-
	TFHE	137.44	0.63	1,578	0.58	1,729	**236.97**×

with matrix multiplication, and N_2-point NTT. Our first optimization involves caching pre-computed constants in SMEM, yields a 5% performance improvement. The second optimization applies our carefully designed data layout for conflict-free memory access, resulting in a 20-30% speedup. Finally, we fuse the two kernels into a single optimized kernel, achieving approximately 50% speedup across all polynomial sizes. These progressive optimizations enhance both compute and memory throughput, maximizing overall performance.

5.2 Performance of Bootstrapping

Table 4 presents performance results for gate bootstrapping of FHEW and TFHE using GD-I/II parameter sets in OpenFHE [ope24] on a 48-thread CPU, bootstrapping of TFHE in MR-I/II from [LLL+24], and our VeloFHE implementation on GPU. Here, we use a batch size of 16,384, and the measured TFHE implementation uses the ternary key version. The execution time is measured in milliseconds (ms), and throughput is expressed in operations per second (OPS). Note that the reported times for VeloFHE include CPU-GPU data transfer latency. Regarding the modulus raising method, as CPU implementation is unavailable, we utilize the data from [LLL+24] for TFHE comparison. Our results demonstrate comparable speedups, achieving microsecond-level amortized bootstrapping times. Under GD-I, we obtain 0.14 ms and 0.09 ms for FHEW and TFHE on RTX 4090 GPU, corresponding to 7,007 and 11,378 OPS throughput, respectively. These results

Table 5: Execution time of homomorphic functions on RTX 4090 GPU, with comparisons to [XLK+25]. Here, "T" and "B" denote the ternary and binary key TFHE implementations in VeloFHE, resepctively. The speedup is calculated using the ternary version.

Functions	Paramters			Execution time (ms)			Speedup
	n	q	N	[XLK+25]	VeloFHE-T	VeloFHE-B	
EvalBinGate	512	2^{10}	2^{10}	0.23	0.09	0.08	**2.56**×
EvalFunc	1,305	2^{11}	2^{11}	1.59	0.71	0.69	**2.24**×
EvalFloor	1,305	2^{11}	2^{10}	1.68	0.72	0.70	**2.33**×

Table 6: Comparative analysis of FHE bootstrapping acceleration across various platforms. Here, λ refers to security level.

	Scheme	Parameters (N, λ)	Time (ms)	Throughput (OPS)	Type	Methodology
[GBA24]	TFHE	$(2^{11}, 127)$	29.85	33	AVX512	Intel Xeon Platinum 8252C
[BBTV23]	FHEW	$(2^{10}, 128)$	18.5	54	FPGA	Xilinx Alveo U280
[YKP22]	TFHE	$(2^{10}, 80)$	3.76	3,454	FPGA	Xilinx Virtex UltraScale+ VU13P
		$(2^{11}, 128)$	9.56	1,150		
[BDTV23]	TFHE	$(2^{10}, 80)$	0.74	25,000	FPGA	Xilinx Alveo U280
[MBS24]	TFHE	$(2^{10}, 128)$	1.6	625	FPGA	Xilinx Alveo U280
[KL24]	TFHE	$(2^{10}, 80)$	1.43	699	FPGA	Xilinx Virtex Ultrascale+ VCU128
		$(2^{10}, 110)$	2.35	425		
[JLJ22]	TFHE	$(2^{10}, 80)$	0.2	10,000	ASIC	TSMC 28nm 1.2GHZ
[PPC+23]	TFHE	$(2^{10}, 110)$	0.23	39,600	ASIC	TSMC 28nm 1.2GHz
		$(2^{11}, 128)$	0.44	21,104		
[PPK24]	TFHE	$(2^{10}, 110)$	0.2	78,692	ASIC	TSMC 28nm 1.2GHz
		$(2^{11}, 128)$	0.38	41,850		
[DFH+24]	TFHE	$(2^{10}, 110)$	-	85,034	ASIC	TSMC 7nm 1GHz
		$(2^{11}, 128)$	-	45,246		
[cuf24]	TFHE	$(2^{10}, 110)$	0.18	5,555	GPU	NVIDIA A100
[XLK+25]	TFHE	$(2^{10}, 128)$	0.23	4,347	GPU	NVIDIA RTX 4090
VeloFHE	TFHE	$(2^{10}, 110)$	0.13	7,501	GPU	NVIDIA A100
		$(2^{10}, 128)$	0.09	11,378	GPU	NVIDIA RTX 4090
		$(2^{10}, 110)$	**0.08**	**11,779**		

represent 371.86× and 390.44× speedups over 48-thread CPU execution.

Comparisons with Related Works. In Table 5, we compare our implementation with the state-of-the-art GPU approach [XLK+25]. We evaluate three homomorphic functions, including gate bootstrapping, homomorphic evaluation of arbitrary functions, and homomorphic flooring operation, denoted as EvalBinGate, EvalFunc, and EvalFloor, respectively. These functions are provided by the OpenFHE library [ope24], which is instantiated using TFHE. To ensure a fair comparison, we follow the settings in [XLK+25] and use a same batch size of 16,384 on the same GPU device. The work in [XLK+25] employs FFT and the cooperative kernel technique to implement a multi-block design. However, it relies on time-consuming grid-level synchronization, and offers few advantage for batched operations. In contrast, our approach avoids these limitations and allows flexible thread block configurations. Under the same conditions, our implementation achieves speedups of 2.56×, 2.24×, and 2.33× for EvalBinGate, EvalFunc, and EvalFloor, respectively, demonstrating the effectiveness of our design and optimizations.

Table 6 presents a comparison of our approach with existing studies on accelerating

TFHE across various hardware platforms. Here, λ denotes the security level. MOSFHET [GBA24] employs AVX512 instructions, but can offer relatively low parallelism and struggles to achieve high throughput. Custom ASICs generally offer higher throughput than FPGAs. Simulation results for leading ASIC designs such as MATCHA [JLJ22], Strix [PPC+23], and Morphling [PPK24] show performance reaching the ten-million OPS range. On the GPU side, cuFHE [cuf24], which focuses on TFHE gate bootstrapping with fixed $N = 1024$ and binary secret keys at a 110-bit security level, delivers 5,555 OPS on an A100 GPU. Another recent GPU-based implementation [XLK+25] achieves a throughput of 4,347 OPS on an RTX 4090 GPU under a 128-bit security setting. Our method provides a 35.03% performance improvement over cuFHE [cuf24] when both use the same binary key distribution. Under the same parameters and on the same platform as [XLK+25], our approach delivers a 2.62× speedup. More specifically, our TFHE gate bootstrapping design at $N = 1024$ achieves 11,378 and 11,779 OPS throughput on an RTX 4090 GPU. This level of performance places our solution on par with some hardware accelerators in terms of throughput, highlighting the efficiency and scalability of our GPU-based approach.

6 Conclusion

In this work, we present VeloFHE, a CUDA-accelerated design that enhances FHEW and TFHE homomorphic encryption performance on GPUs. Our novel hybrid four-step NTT for polynomial multiplication and optimizations in blind rotation and key switching address computational bottlenecks. GPU-specific enhancements, including warp-level programming and efficient memory utilization, yield substantial speedups over existing implementations. These advancements demonstrate the viability of GPU-accelerated homomorphic encryption, facilitating practical privacy-preserving computations for large ciphertexts. Future research may extend these optimizations to other schemes and emerging GPU architectures.

Acknowledgements

We would like to thank the anonymous reviewers for their careful reading of our manuscript and their insightful comments. This work is supported in part by the National Natural Science Foundation of China (62132008, 62372417, and U22B2030), the Natural Science Foundation of Jiangsu Province (BK20220075), the Innovation and Technology Fund (ITS/098/22), the Shanghai Collaborative Innovation Fund (XTCX-KJ-2023-54), the Special Fund for Key Technologies in Blockchain of Shanghai Scientific and Technological Committee (23511100300), and the CityUHK Research Matching Grant Scheme (9239077 and 9239083).

References

[ACC+21] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, pages 31–62, 2021.

[AdCY+23] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. FAB: an fpga-based accelerator for bootstrappable fully homomorphic encryption. In *IEEE International Symposium on High-Performance Computer*

Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023, pages 882–895. IEEE, 2023.

[AMK+23] Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. REED: chiplet-based scalable hardware accelerator for fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 1190, 2023.

[AP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314. Springer, 2014.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[Bai90] David H. Bailey. FFTs in external or hierarchical memory. *J. Supercomput.*, 4(1):23–35, 1990.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BBTV23] Jonas Bertels, Michiel Van Beirendonck, Furkan Turan, and Ingrid Verbauwhede. Hardware acceleration of FHEW. In Maksim Jenihhin, Hana Kubátová, Nele Metens, Jaan Raik, Foisal Ahmed, and Jan Belohoubek, editors, *26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2023, Tallinn, Estonia, May 3-5, 2023*, pages 57–60. IEEE, 2023.

[BDF18] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. Large FHE gates from tensored homomorphic accumulator. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 217–251. Springer, 2018.

[BDTV23] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 741–755. ACM, 2023.

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

[BIP+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022,*

Proceedings, Part II, volume 13792 of *Lecture Notes in Computer Science*, pages 188–215. Springer, 2022.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512. Springer, 2018.

[BPA+21] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. *IEEE Trans. Emerg. Top. Comput.*, 9(2):941–956, 2021.

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

[BVMA18] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance FV somewhat homomorphic encryption on gpus: An implementation using CUDA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):70–95, 2018.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, 2016.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In Mitsuru Matsui, editor, *Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings*, volume 11405 of *Lecture Notes in Computer Science*, pages 106–126. Springer, 2019.

[CJL+20] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE. In *WAHC 2020 - 8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, [Virtual], France, December 2020.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*, volume 12716 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2021.

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

[cuf24] cuFHE: CUDA-Accelerated Fully Homomorphic Encryption Library. https://github.com/vernamlab/cuFHE, October 2024.

[DFH+24] Xianglong Deng, Shengyu Fan, Zhicheng Hu, Zhuoyu Tian, Zihao Yang, Jiangrui Yu, Dingyuan Cao, Dan Meng, Rui Hou, Meng Li, Qian Lou, and Mingzhe Zhang. Trinity: A general purpose FHE accelerator. In *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*, pages 338–351. IEEE, 2024.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

[fhe24] FHEW: A Fully Homomorphic Encryption library. https://github.com/lducas/FHEW, October 2024.

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

[FWX+23] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. Tensorfhe: Achieving practical computation on encrypted data using GPGPU. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 922–934. IEEE, 2023.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):229–253, 2021.

[GBA24] Antonio Guimarães, Edson Borin, and Diego F. Aranha. MOSFHET: optimized software for FHE over the torus. *J. Cryptogr. Eng.*, 14(3):577–593, 2024.

[GBP+23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.

[GINX16] Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 528–558. Springer, 2016.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[heo24] HEonGPU: A GPU based Homomorphic Encryption Library. https://github.com/Alisah-Ozcan/HEonGPU, December 2024.

[HP23] Daniel Heinz and Thomas Pöppelmann. Combined fault and DPA protection for lattice-based cryptography. *IEEE Trans. Computers*, 72(4):1055–1066, 2023.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):114–148, 2021.

[JLJ22] Lei Jiang, Qian Lou, and Nrushad Joshi. MATCHA: a fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 235–240. ACM, 2022.

[KKC+23] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In Yan Solihin and Mark A. Heinrich, editors, *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, pages 18:1–18:15. ACM, 2023.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 711–725. ACM, 2022.

[KL24] Tianqi Kong and Shuguo Li. Hardware acceleration and implementation of fully homomorphic encryption over the torus. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 71(3):1116–1129, 2024.

[KLK+22] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. ARK: fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, pages 1237–1254. IEEE, 2022.

[KS23] Kamil Kluczniak and Leonard Schild. FDFB: full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):501–537, 2023.

[LGZ+19] Xinya Lei, Ruixin Guo, Feng Zhang, Lizhe Wang, Rui Xu, and Guangzhi Qu. Accelerating homomorphic full adder based on FHEW using multicore CPU and gpus. In Zheng Xiao, Laurence T. Yang, Pavan Balaji, Tao Li, Keqin Li, and Albert Y. Zomaya, editors, *21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12, 2019*, pages 2508–2513. IEEE, 2019.

[LLCP15] Moon Sung Lee, Yongje Lee, Jung Hee Cheon, and Yunheung Paek. Accelerating bootstrapping in FHEW using gpus. In *26th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2015, Toronto, ON, Canada, July 27-29, 2015*, pages 128–135. IEEE Computer Society, 2015.

[LLL+24] Zhihao Li, Ying Liu, Xianhui Lu, Ruida Wang, Benqiang Wei, Chunling Chen, and Kunpeng Wang. Faster bootstrapping via modulus raising and composite NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):563–591, 2024.

[LMK+23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 227–256. Springer, 2023.

[MAK+23] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. Medha: Microcoded hardware accelerator for computing on encrypted data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):463–500, 2023.

[MAM20] Toufique Morshed, Md Momin Al Aziz, and Noman Mohammed. CPU and GPU accelerated fully homomorphic encryption. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 142–153. IEEE, 2020.

[Mat20] Kotaro Matsuoka. TFHEpp: pure C++ implementation of TFHE cryptosystem. https://github.com/virtualsecureplatform/TFHEpp, 2020.

[MBS24] Kotaro Matsuoka, Song Bian, and Takashi Sato. HOGE: homomorphic gate on an FPGA. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference, ASPDAC 2024, Incheon, Korea, January 22-25, 2024*, pages 325–332. IEEE, 2024.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MP21] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in fhew-like cryptosystems. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*, pages 17–28. WAHC@ACM, 2021.

[nuf] nuFHE: A GPU implementation of fully homomorphic encryption on torus, howpublished = https://github.com/nucypher/nufhe, month = dec, year = 2024.

[ÖEM+22] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. Efficient number theoretic transform implementation on GPU for homomorphic encryption. *J. Supercomput.*, 78(2):2840–2872, 2022.

[ope24] OpenFHE: Open-Source Fully Homomorphic Encryption Library. https://github.com/openfheorg/openfhe-development, October 2024.

[ÖS23] Ali Sah Özcan and Erkay Savas. Two algorithms for fast GPU implementation of NTT. *IACR Cryptol. ePrint Arch.*, page 1410, 2023.

[ÖS24] Ali Sah Özcan and Erkay Savas. Heongpu: a gpu-based fully homomorphic encryption library 1.0. *IACR Cryptol. ePrint Arch.*, page 1543, 2024.

[pha24] PhantomFHE: A CUDA-Accelerated Fully Homomorphic Encryption Library. https://github.com/encryptorion-lab/phantom-fhe, October 2024.

[PPC+23] Adiwena Putra, Prasetiyo, Yi Chen, John Kim, and Joo-Young Kim. Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*, pages 1319–1331. ACM, 2023.

[PPK24] Prasetiyo, Adiwena Putra, and Joo-Young Kim. Morphling: A throughput-maximized tfhe-based accelerator using transform-domain reuse. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*, pages 249–262. IEEE, 2024.

[RCK+21] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 26–39. IEEE, 2021.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[RJV+18] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. Hepcloud: An fpga-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Trans. Computers*, 67(11):1637–1650, 2018.

[RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.

[RTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 387–398. IEEE, 2019.

[S+01] Victor Shoup et al. NTL: A library for doing number theory. 2001.

[SFK+21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*, pages 238–252. ACM, 2021.

[SYD+24] Shiyu Shen, Hao Yang, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. Leveraging gpu in homomorphic encryption: Framework design and analysis of bfv variants. *IEEE Transactions on Computers*, pages 1–12, 2024.

[SYL+23] Shiyu Shen, Hao Yang, Yu Liu, Zhe Liu, and Yunlei Zhao. CARM: cuda-accelerated RNS multiplication in word-wise homomorphic encryption schemes for internet of things. *IEEE Trans. Computers*, 72(7):1999–2010, 2023.

[TCBS23a] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. Building blocks for LSTM homomorphic evaluation with TFHE. In Shlomi Dolev, Ehud Gudes, and Pascal Paillier, editors, *Cyber Security, Cryptology, and Machine Learning - 7th International Symposium, CSCML 2023, Be'er Sheva, Israel, June 29-30, 2023, Proceedings*, volume 13914 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2023.

[TCBS23b] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 79–90. ACM, 2023.

[TÖA+22] Enes Recep Türkoglu, Ali Sah Özcan, Can Ayduman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. An accelerated GPU library for homomorphic encryption operations of BFV scheme. In *IEEE International Symposium on Circuits and Systems, ISCAS 2022, Austin, TX, USA, May 27 - June 1, 2022*, pages 1155–1159. IEEE, 2022.

[WWL+24] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: Faster and smaller. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 342–372. Springer, 2024.

[XLK+25] Yu Xiao, Feng-Hao Liu, Yu-Te Ku, Ming-Chien Ho, Chih-Fan Hsu, Ming-Ching Chang, Shih-Hao Hung, and Wei-Chao Chen. Gpu acceleration for fhew/tfhe bootstrapping. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):314–339, 2025.

[XZD+23] Binwu Xiang, Jiang Zhang, Yi Deng, Yiran Dai, and Dengguo Feng. Fast blind rotation for bootstrapping fhes. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 3–36. Springer, 2023.

[YKP22] Tian Ye, Rajgopal Kannan, and Viktor K. Prasanna. FPGA acceleration of fully homomorphic encryption over the torus. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*, pages 1–7. IEEE, 2022.

[YSD+24] Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. Phantom: A cuda-accelerated word-wise homomorphic encryption library. *IEEE Trans. Dependable Secur. Comput.*, 21(5):4895–4906, 2024.

[Zam22] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.

[ZCL+24] Mengxin Zheng, Cheng Chu, Qian Lou, Nathan Youngblood, Mo Li, Sajjad Moazeni, and Lei Jiang. OFHE: an electro-optical accelerator for discretized TFHE. In Pascal Meinerzhagen, Kapil Dev, and Jerald Yoo, editors, *Proceedings of the 29th ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED 2024, Newport Beach, CA, USA, August 5-7, 2024*, pages 1–6. ACM, 2024.

[ZNW+24] Minxuan Zhou, Yujin Nam, Xuan Wang, Youhak Lee, Chris Wilkerson, Raghavan Kumar, Sachin Taneja, Sanu Mathew, Rosario Cammarota, and Tajana Rosing. UFC: A unified accelerator for fully homomorphic encryption. In *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*, pages 352–365. IEEE, 2024.

[ZZF+24] Tian Zhou, Fangyu Zheng, Guang Fan, Lipeng Wan, Wenxu Tang, Yixuan Song, Yi Bian, and Jingqiang Lin. Convkyber: Unleashing the power of AI accelerators for faster kyber with novel iteration-based approaches. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):25–63, 2024.

A Modular Raising in TFHE Bootstrapping

We present the original proposal for employing the modulus raising technique in TFHE bootstrapping, as introduced in [LLL+24]. For a ternary LWE secret key $\mathbf{s} \in \{-1, 0, 1\}$, the bootstrapping key is generated as follows:

$$\mathbf{bsk} = \begin{cases} \mathbf{bsk}_{i,0} = \mathsf{RGSW}^{N,PQ}_{\mathbf{s}'}(0), \mathbf{bsk}_{i,1} = \mathsf{RGSW}^{N,PQ}_{\mathbf{s}'}(1), \text{ if } (s_i = -1)\,; \\ \mathbf{bsk}_{i,0} = \mathsf{RGSW}^{N,PQ}_{\mathbf{s}'}(0), \mathbf{bsk}_{i,1} = \mathsf{RGSW}^{N,PQ}_{\mathbf{s}'}(0), \text{ if } (s_i = 0)\,; \\ \mathbf{bsk}_{i,0} = \mathsf{RGSW}^{N,PQ}_{\mathbf{s}'}(1), \mathbf{bsk}_{i,1} = \mathsf{RGSW}^{N,PQ}_{\mathbf{s}'}(0), \text{ if } (s_i = 1)\,. \end{cases}$$

The complete GINX-type blind rotation using modulus raising for TFHE is provided in Algorithm 4. Here, the accumulation is performed over $\mathcal{R}_{PQ}$, and the scaling operation is

Algorithm 4 GINX Blind Rotation with Modulus Raising and Composite NTT [LLL+24]

Input: LWE ciphertext $\mathbf{ct} = (\mathbf{a}, b) \in \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, bootstrapping key $\mathbf{bsk}$.

Output: LWE ciphertext $\mathbf{acc} \in \mathsf{RLWE}_{\mathbf{s}'}^{N,Q}(X^{b-\sum_{i=0}^{n-1} a_i s_i})$

1: Set $\mathbf{acc}_0 = (\mathbf{0}, X^b) \in \mathcal{R}_Q^2$
2: **for** $i \in [1, n]$ **do**
3: $\quad \widehat{\mathbf{acc}}_i \leftarrow \mathsf{Com\text{-}NTT}(\mathbf{acc}_i)$ ▷ Composite NTT
4: $\quad \widehat{\mathbf{acc}}_i \leftarrow \widehat{\mathbf{acc}}_i \odot (\mathbf{1} + (X^{a_i} - 1) \odot \mathbf{bsk}_{i,0} + (X^{-a_i} - 1) \odot \mathbf{bsk}_{i,1}) \in \mathcal{R}_{PQ}^2$ ▷ External product
5: $\quad \mathbf{acc}_i \leftarrow \mathsf{Com\text{-}iNTT}(\widehat{\mathbf{acc}}_i)$ ▷ Composite inverse NTT
6: $\quad \mathbf{acc}_i \leftarrow \lfloor \mathbf{acc}_i / P \rceil \in \mathcal{R}_Q^2$ ▷ Scaling by P
7: **return** $\mathbf{acc}_n$

applied to $\mathbf{acc}_i$. For efficient bootstrapping computation, the keys should be precomputed and stored in NTT form in $\mathcal{R}_{PQ}$. When using unsigned integer types, the representation of negative values must be adapted. This may introduce compatibility challenges among variants that employ different external product techniques, particularly when developing a unified framework accommodating multiple schemes and variants.

B Modular Reduction in CUDA PTX Assembly

Efficient modular reduction is fundamental to many cryptographic algorithms. Traditional methods of computing $a \bmod P$ via division can be prohibitively expensive. Consequently, several techniques have been developed to accelerate modular reduction by exploiting algebraic or architectural properties. Below we introduce four commonly employed methods, including special reduction using Goldilocks primes, Barrett reduction [Bar86], Montgomery reduction [Mon85], and Shoup-type reduction [S+01] using constant precomputation.

Algorithm 5 Modular reduction in cuFHE [cuf24]

Input: A 64-bit integer a, a Goldilocks prime moduli $P = 2^{64} - 2^{32} + 1$
Output: A 64-bit result of $a \bmod P$

```
.reg.u32   m          ▷ Declare a 32-bit register m
.reg.u64   t          ▷ Declare a 64-bit register t
set.ge.u32.u64  m, a, P   ▷ Set m = 1 if a ≥ P, else m = 0
mov.b64    t, {m, 0}      ▷ Move m into 64-bit register t
add.u64    a, a, t        ▷ Conditionally add t to a
```

return a

Algorithm 6 Barrett modular reduction implementation in Phantom [pha24]

Input: $a = 2^{64} \cdot a_1 + a_0$, $\gamma = 2^{64} \cdot \gamma_1 + \gamma_0$, q, $0 \le a, \gamma, q < 2^{64}$
Output: $res = a \bmod q, 0 \le res < 2q$

```
mul.hi.u64   tmp, a0, γ0
mad.lo.cc.u64  tmp, a0, γ1, tmp   ▷ c0 ← ⌊((a0γ0)hi + (a0γ1)lo) / 2^64⌋
madc.hi.u64  res, a0, γ1, 0
mad.lo.cc.u64  tmp, a1, γ0, tmp   ▷ c1 ← ⌊((a0γ0)hi + (a0γ1)lo + (a1γ0)lo) / 2^64⌋
madc.hi.u64  res, a1, γ0, res
mad.lo.u64   res, a1, γ1, res     ▷ res ← (a0γ1)hi + (a1γ0)hi + (a1γ1)lo
mul.lo.u64   res, res, q
sub.u64      res, a0, res         ▷ res ← a0 − (res · q)lo
sub.s64      res, res, q
shr.s64      tmp, res, 63
and.b64      tmp, tmp, q
add.s64      res, res, tmp
```

return res

Algorithm 7 Implementation of Montgomery reduction in VeloFHE

Input: 64-bit integers a_1 and a_0, where $a = a_1 2^{64} + a_0$, modulus P, and Montgomery constant μ
Output: a 64-bit result $r \equiv aR^{-1} \bmod P$

```
1:   .reg.u64  t                          ▷ Declare a 64-bit register t
2: mul.lo.u64  t, a0, μ                   ▷ Store the low 64-bit of a0 · μ in t
3: mul.hi.u64  t, t, P                    ▷ Store the high 64-bit of t · P in t
4:    add.u64  r, a1, P                   ▷ r ← a1 + P
5:    sub.u64  r, r, t                    ▷ r ← r − t
6: return r
```

Algorithm 8 Shoup-type modular multiplication implementation in Phantom [pha24]

Input: 64-bit integers x and y, a Shoup-type constant γ', modulus P
Output: a 64-bit result $a \equiv x \cdot y \bmod P$

```
1: mul.hi.u64  t1, x, γ'                  ▷ Store the high 64-bit of xγ' in t1
2: mul.lo.u64  t2, x, y                   ▷ Store the low 64-bit of xy in t2
3: mul.lo.u64  t3, t1, P                  ▷ Multiply t1 with P and store the low 64-bit result in t3
4:    sub.u64  a, t2, t3                  ▷ Subtract t3 from t2
5: return a
```

The cuFHE library [cuf24] uses a Goldilocks prime $P = 2^{64} - 2^{32} + 1$ to enable efficient modular reduction via bit masking, as outlined in Algorithm 5. For a 64-bit integer a, the procedure first compares a with P to determine the value of m, then conditionally applies bit masking to yield the reduced result. Notably, the `set` instruction depends on a comparison, causing the GPU to split execution paths internally when handling conditional additions. As a result, threads follow different control-flow paths, which can induce divergence in warp execution.

For modular multiplication, it is often necessary to reduce a 128-bit product constructed using 64-bit instructions for GPU implementation. A widely adopted technique for modular multiplication of two integers is Barrett reduction [Bar86], which replaces explicit division by P with multiplication by a precomputed constant. Let $a = x \cdot y$ be the 128-bit product of two 64-bit integers x and y. Define $\beta = 2^{\ell}$ as the base, where ℓ is typically the word length, and compute $\gamma = \lfloor \beta / P \rfloor$. To obtain $a \bmod P$, one subtracts P multiple of $\lfloor a\gamma/\beta \rfloor$ from a. Algorithm 6 provides an implementation from the Phantom library [pha24]. Due to CUDA's computational constraints, the algorithm processes the 128-bit intermediate value by separately handling the high and low 64-bit parts. This design preserves constant thread workloads by implicitly recording overflow flags in the condition code register and correctly managing carry bits in additions.

Despite its efficiency, the previous approach still requires many instructions. A more compact solution that uses fewer instructions is therefore preferable. As a result, we employ Montgomery reduction [Mon85] in the blind rotation. Montgomery reduction transforms computations modulo P into computations involving a representation that is more conducive to efficient multiplication and reduction. In the Montgomery domain, an integer a is represented as $a' = aR \bmod P$. We also precompute $\mu = P \bmod R$, so the multiplication result can be reduced via $(a\mu \bmod R)P - \lfloor a\mu/R \rfloor$. This method requires fewer multiplications than Barrett reduction. Our optimized CUDA PTX assembly implementation is presented in Algorithm 7, where we set $R = 2^{64}$. Note that we add P to a_1 to prevent overflow during unsigned subtraction.

Another widely adopted approach is Shoup-type modular multiplication [S+01], which assumes one multiplier is constant. In this method, we can precompute $\gamma' = \lfloor y\beta/P \rfloor$ to simplify the computation of $\lfloor x \cdot y\gamma/\beta \rfloor$ in Barrett reduction. Algorithm 8 presents an implementation of this technique in the Phantom library [pha24]. It is commonly employed in NTT, where a table of Shoup-type constants for twiddle factors is precomputed to facilitate modular multiplication. Although it reduces the instruction count, this method doubles memory usage on GPUs because both twiddle factors and Shoup-type constants

must be stored in SMEM. This often precludes further kernel fusion, which may reduce SM occupancy due to high SMEM demands and ultimately degrade performance. Consequently, using Montgomery reduction at this stage can reduce SMEM usage, which is advantageous for fused kernels with NTT.