BOLT-FHE: An Efficient Unified Framework for GPU-based TFHE Bootstrapping via On-Chip Local Tiling Strategies

Yanren Chen[1], Fangyu Zheng[2(✉)], Guang Fan[2], Jiankuo Dong[1], Wenxu Tang[1], Tian Zhou[1], Jingqiang Lin[1] and Jiwu Jing[2]

[1] School of Cyber Science and Technology, University of Science and Technology of China, Hefei, China
yanrenchen@mail.ustc.edu.cn,djiankuo@gmail.com,wenxutang@mail.ustc.edu.cn, weekdayzt@mail.ustc.edu.cn,linjq@ustc.edu.cn

[2] School of Cryptology, University of Chinese Academy of Sciences, Beijing, China
zhengfangyu@ucas.ac.cn,fanguang328@gmail.com,jwjing@ucas.ac.cn

Abstract. Bootstrapping is the main performance bottleneck in bitwise Fully Homomorphic Encryption (FHE), and practical acceleration requires careful orchestration of the blind rotation and external product chain under GPU resource constraints. This paper presents BOLT-FHE, a GPU bootstrapping framework that emphasizes block-local execution, on-chip tiling, and a unified MegaKernel supporting both gadget decomposition and modulus raising, with optional support for a recently proposed technique (Bergerat *et al.*, CHES 2025) based on the common mask assumption (CM packing). Our design keeps the accumulator update chain within a single thread block and fuses NTT/INTT, external products, and accumulator updates using a fixed execution template. Two compile-time parameters—WPP (warps per polynomial) and IPT (items per thread)—control multi-warp cooperation and per-thread register footprint, enabling consistent kernel structure across different parameter sets.
On an NVIDIA RTX 4090, BOLT-FHE reaches 40,166 bootstrappings per second at 128-bit security, demonstrating high-throughput TFHE bootstrapping on a commodity GPU. Compared to the state-of-the-art GPU implementation VeloFHE (Shen *et al.*, CHES 2025), BOLT-FHE achieves 1.85×–2.92× speedups with gadget decomposition. In particular, for modulus raising, BOLT-FHE improves by 2.38×–2.42× without CM packing, and by 3.17×–3.31× under the best packing configuration, reflecting the combined benefits of fused arithmetic, more regular memory access, and amortization enabled by CM packing. Overall, BOLT-FHE shows that a portable, fused-kernel organization with explicit on-chip budgeting can substantially improve TFHE bootstrapping throughput while remaining compatible with both noise management paths.

Keywords: Fully Homomorphic Encryption · Bootstrapping · TFHE · External Product · GPU Acceleration

1 Introduction

Fully Homomorphic Encryption (FHE) has long been regarded as the "Holy Grail" of cryptography [Gen09], offering the capability to perform arbitrary computations directly

Revision note. This manuscript is a minor revision of the version published in *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3), pp. 198–223, DOI: 10.46586/tches.v2026.i3.198-223. This revision updates the implementation results for the `uint32`-compatible G1, G3, and G5 parameter sets and adds the open-source implementation link.

Licensed under Creative Commons License CC-BY 4.0.

on encrypted data without decryption. This primitive paves the way for privacy-preserving applications [Zam22, AB24, FGT21] in cloud computing, ranging from secure machine learning to genomic analysis. Since Gentry's breakthrough, several mainstream FHE schemes have emerged to address different computational needs: BGV [BGV12] and BFV [FV12] for exact integer arithmetic, CKKS [CKKS17] for approximate arithmetic in data science, and FHEW [DM15] / TFHE [CGGI20] for fast gate-level operations.

However, practical deployment is fundamentally constrained by the accumulation of cryptographic noise during homomorphic operations. To support unbounded computations, the noise must be periodically reduced via a computationally expensive procedure known as bootstrapping. To bridge the performance gap between plaintext and ciphertext computations, hardware acceleration [JKA+21, FWX+23, FZZ+25, JDW+25] has become an indispensable research direction. Among various platforms, Graphics Processing Units (GPUs) are particularly favored for their massive parallelism and high memory bandwidth, holding a unique position in lattice cryptography acceleration [ZZF+24, ZZX+25].

1.1 Limitations of Prior GPU-based Works

Existing GPU-based implementations [Zam24, XLK+25] for bitwise FHE typically revolve around the orchestration of the external product update chain. For instance, the CUDA backend of TFHE-rs [Zam24] uses a host-side iteration over the LWE dimension combined with a split kernel execution model for the blind rotation. The step-one kernel is responsible for gadget decomposition (GD) and FFT, while the step-two kernel executes the multiply-accumulate operations between accumulator slices and bootstrapping keys, followed by an IFFT. While this design provides structural clarity and simplifies resource management by allowing latency hiding through increased block concurrency, it is unable to preserve architectural state across kernel boundaries. Consequently, the accumulator and intermediate results must be exchanged via GMEM between phases.

In contrast, [XLK+25] proposed a scalable approach that explicitly partitions blind rotation computations across multiple thread blocks. By leveraging `cooperative_groups` for grid-wide synchronization, their method circumvents the hardware resource constraints of a single thread block, enabling support for larger parameter sets.

However, both architectural paradigms lead to a significant structural consequence: inter-phase (cross-kernel) or inter-block communication must rely on globally visible intermediate buffers. Since blind rotation requires n LWE dimension updates, and GD further introduces d_g decomposition levels, such designs cause the off-chip traffic related to the ACC to scale linearly with n, or even with $n \cdot d_g$. This memory bottleneck, coupled with synchronization overhead, prevents these implementations from fully exploiting the high temporal locality of the ACC during the blind rotation loop.

To illustrate this point, we performed a comparative analysis using NVIDIA's profiling tools on the open-source TFHE-rs library. Table 1 presents the profiling results, comparing the TFHE-rs classical PBS parameters `V1_1_PARAM_MESSAGE_2_CARRY_2_KS_PBS_TUNIFORM_2M128` against our proposed BOLT-FHE implementation with `OpenFHE STD128Q3` on the batch size of 16,384. It is important to note that, given the divergence in arithmetic paths (e.g., FFT/floating-point vs. NTT/integer), the primary focus of Table 1 is to highlight the reduction in DRAM traffic and improved data locality achieved by our residency strategy, rather than comparing wall-clock execution time.

1.2 Contributions

Based on this observation, we present BOLT-FHE, an architecture-aware acceleration framework for faster GPU-based **B**ootstrapping via **O**n-chip **L**ocal **T**iling. Our contributions are summarized as follows:

Table 1: Memory Behavior Profiling of Blind Rotation via Nsight Compute

Metric	TFHE-rs CUDA backend		**Ours**
	Step 1	**Step 2**	
Total DRAM Traffic	2,900 GB		11 GB
DRAM Throughput (%)	36.4	47.0	0.6
L2 Hit Rate (%)	66.5	57.0	99.4
L1 Hit Rate (%)	29.3	1.7	14.9
L2 Throughput (%)	12.2	18.0	22.6
L1 Throughput (%)	13.2	9.2	50.9

- **Unified on-chip external product framework across GD/MR with a fixed workspace formulation.** We develop a unified GPU framework for generic GLWE × GGSW external products that supports both gadget decomposition and modulus raising. To prevent the on-chip workspace from scaling with the GD depth, we introduce a staged digit-serial evaluation that rewrites the conventional workflow into a fixed workspace form. As a result, the ACC remains on chip rather than being materialized in global memory, enabling stable execution across a broad range of parameter sets.

- **Throughput scaling via CM packing under controlled on-chip budgets.** Building upon the stabilized on-chip workspace, we incorporate Common Mask (CM) packing to amortize blind rotation key usage across multiple accumulators. We show how GD, MR and CM can be mapped into a single kernel structure with predictable on-chip resource usage, and we characterize the feasible packing regimes and their throughput optima under the coupled SMEM/register constraints.

- **A fused MegaKernel implementation with specialized primitives and comprehensive evaluation.** We present an end-to-end implementation of blind rotation that sustains on-chip residency across the full arithmetic pipeline. Our design includes (i) a warp-level organization with explicit knobs (WPP/IPT) for synchronization granularity and register budgeting, (ii) a packed radix-2 sub-warp NTT tailored for in-SMEM transforms, and (iii) a MegaKernel organization that significantly reduces store-reload boundaries. We evaluate the resulting system across representative GD and MR parameter sets, with and without CM packing.

We conduct a broad evaluation across both the conventional GD approach and the recently proposed MR approach [LLL+24], and further fine-tune the MR+CM combination [WLX+25] to identify its best performing configurations. Compared to the state-of-the-art GPU implementation VeloFHE [SYL+25], BOLT-FHE achieves 2.38×–2.42× speedups on the MR approach without CM, and improves to 3.17×–3.31× after enabling CM and searching for the optimal packing configuration. For the traditional GD approach, BOLT-FHE also delivers consistent gains ranging from 1.85× to 2.92×. Notably, under a 128-bit security parameter set, we report a peak throughput of **40,166 gate bootstrappings per second** on a commodity GPU (NVIDIA RTX 4090), demonstrating the practical advantage of a throughput-driven GPU design.

2 Preliminaries

2.1 Notations

We denote the set of integers by $\mathbb{Z}$, real numbers by $\mathbb{R}$. For a positive integer Q, we identify $\mathbb{Z}_Q$ with the set of integers in $(-Q/2, Q/2] \cap \mathbb{Z}$. We use bold lower-case letters (e.g., $\mathbf{a}$) to

denote vectors and bold upper-case letters (e.g., $\mathbf{A}$) to denote matrices. The inner product of two vectors is denoted by $\langle \mathbf{a}, \mathbf{b} \rangle$. In the absence of CM packing, we set $r = 1$ by default.

Table 2: Notations used in this work

Symbol	Description
n	Dimension of LWE ciphertext
r	Common Mask packing factor
q	Modulus of LWE ciphertexts
Q	Modulus of GLWE ciphertexts
T	Auxiliary raised modulus in MR ($T > Q$)
N	Polynomial degree in $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$
k	Rank of GLWE
χ	Error distribution
$q_{\mathrm{KS}}, d_{\mathrm{KS}}$	Modulus and decomposition length for Key Switch
B_g, d_g	Base and decomposition length for gadget decomposition
$\mathbf{S}$	Secret key for CM-GGSW/GLWE, $\mathbf{S} \in \mathcal{R}_Q^{r \times k}$
$\vec{s}$	Secret key $(s_1, \ldots, s_r)$ for CM-LWE, $s_j = (s_{1j}, \ldots, s_{nj})^T$

Polynomial Rings. We define the cyclotomic polynomial ring as $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where N is a power of two. We denote $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R} = \mathbb{Z}_Q[X]/(X^N + 1)$ as the ring of polynomials with coefficients in $\mathbb{Z}_Q$. Operations on polynomials (multiplication, addition) are performed modulo $X^N + 1$.

General LWE (GLWE). Let $k \geq 1$ be an integer and $\mathbf{S} \in \mathcal{R}_Q^{r \times k}$ be the secret key. When $r = 1$, we identify $\mathbf{S}$ with a vector in $\mathcal{R}_Q^k$. A GLWE ciphertext of a message $m \in \mathcal{R}_Q$ is a tuple $\mathbf{c} = (\mathbf{a}, b) \in \mathcal{R}_Q^k \times \mathcal{R}_Q$, where $\mathbf{a} \leftarrow \mathcal{U}(\mathcal{R}_Q^k)$ is uniformly sampled and

$$b = \langle \mathbf{a}, \mathbf{S} \rangle + m + e, \tag{1}$$

with $e \leftarrow \chi$. When $N = 1$, this corresponds to standard LWE [Reg09]; when $k = 1$, it corresponds to RLWE.

GGSW Ciphertexts. General GSW (GGSW) ciphertexts are typically used as bootstrapping keys. Under gadget decomposition, a GGSW ciphertext encrypting $m \in \mathcal{R}_Q$ is a rectangular matrix

$$\mathbf{C}_{\mathrm{GD}} = \mathbf{Z} + m \cdot \mathbf{G} \in \mathcal{R}_Q^{(k+1)d_g \times (k+1)} \tag{2}$$

where $\mathbf{Z}$ is a matrix of GLWE encryptions of 0 (under modulus Q), and $\mathbf{G}$ is the gadget matrix induced by (B_g, d_g):

$$\mathbf{g} \;=\; (1, B_g, \ldots, B_g^{d_g - 1}) \in \mathbb{Z}^{d_g}, \qquad \mathbf{G} \;=\; \mathrm{diag}(\mathbf{g}) \otimes \mathbf{I}_{k+1} \;\in\; \mathcal{R}_Q^{(k+1)d_g \times (k+1)}. \tag{3}$$

In this work, we also consider a variant instantiated via Modulus Raising (MR), a technique that has recently emerged in TFHE/FHEW research [LLL+24, WLX+25]. An MR-based GGSW ciphertext encrypting $m \in \mathcal{R}_Q$ is a compact square matrix

$$\mathbf{C}_{\mathrm{MR}} = \mathbf{Z} + m \cdot \frac{T}{Q} \cdot \mathbf{I}_{k+1} \in \mathcal{R}_T^{(k+1) \times (k+1)} \tag{4}$$

where $\mathbf{Z}$ is a matrix of GLWE encryptions of 0 (under modulus T).

2.2 External Product

The external product, denoted by $\boxdot$, is the computational core of bootstrapping. It homomorphically multiplies a GLWE ciphertext $\mathbf{c} \in \mathrm{GLWE}_{\mathbf{S}}(\mu)$ by a GGSW ciphertext $\mathbf{C}$ encrypting a message m, yielding $\mathbf{c}' \in \mathrm{GLWE}_{\mathbf{S}}(\mu \cdot m)$.

Gadget Decomposition Approach. Let $\mathbf{C}_{\mathrm{GD}} \in \mathcal{R}_Q^{(k+1)d_g \times (k+1)}$. Define the digit decomposition operator under (B_g, d_g) as

$$\mathbf{G}^{-1}(\cdot) \;=\; \mathrm{Decomp}_{B_g,d_g}(\cdot) : \mathcal{R}_Q^{k+1} \to \mathcal{R}_Q^{(k+1)d_g}. \tag{5}$$

The GD external product is

$$\mathbf{c}' \;=\; \mathbf{G}^{-1}(\mathbf{c}) \cdot \mathbf{C}_{\mathrm{GD}} \;\in\; \mathcal{R}_Q^{k+1}. \tag{6}$$

Modulus Raising Approach. Let $\mathbf{C}_{\mathrm{MR}} \in \mathcal{R}_T^{(k+1)\times(k+1)}$. The MR external product is

$$\mathbf{c}' \;=\; \left\lfloor \frac{Q}{T} \cdot (\mathbf{c} \cdot \mathbf{C}_{\mathrm{MR}} \bmod T) \right\rceil \;\in\; \mathcal{R}_Q^{k+1}. \tag{7}$$

2.3 Binary FHE Schemes and Bootstrapping

We focus on the TFHE scheme [CGGI20], instantiated within the unified framework of FHEW-like cryptosystems [MP21]. The core operation is functional bootstrapping, which evaluates a look-up table while reducing noise. The pipeline transforms an input LWE ciphertext $\mathbf{c}_{in} \in \mathrm{LWE}_{\vec{s}}^{n,q}(m)$ into a refreshed ciphertext $\mathbf{c}_{out} \in \mathrm{LWE}_{\vec{s}}^{n,q}(f(m))$ through five sequential building blocks:

1) Initialization. The LWE ciphertext is first mapped to the polynomial domain. Let $v(X) \in \mathcal{R}_Q$ be the polynomial embedding the LUT. The accumulator is initialized as a trivial GLWE ciphertext encrypting the LUT rotated by the approximate payload $\tilde{b}$:

$$\mathrm{ACC}_0 \leftarrow (0, \ldots, 0, X^{-\tilde{b}} \cdot v(X)) \in \mathrm{GLWE}_{\mathbf{S}}^{k,Q}(X^{-\tilde{b}} \cdot v(X)) \tag{8}$$

2) Blind Rotation. This step homomorphically accumulates the decryption of the secret key components s_i into the exponent. It iteratively applies a CMux gate controlled by the bootstrapping key (BSK). The CMux gate, constructed via the external product ($\boxdot$), selects between two inputs d_0, d_1 based on a GGSW control bit C:

$$\mathrm{CMux}(C, d_1, d_0) = C \boxdot (d_1 - d_0) + d_0 \tag{9}$$

3) Sample Extraction. This operation projects the GLWE accumulator back to the LWE. We define it for a general GLWE ciphertext $\mathbf{c} = (\mathbf{a}_1, \ldots, \mathbf{a}_k, b) \in \mathrm{GLWE}_{\mathbf{S}}^{k,Q}(\mu)$. Extracting the constant term yields an LWE ciphertext of dimension $N \cdot k$:

$$\text{Sample Extraction}(\mathbf{c}) = (\phi(\mathbf{a}_1) \| \ldots \| \phi(\mathbf{a}_k), b_0) \in \mathrm{LWE}_{\vec{s}'}^{kN,Q}(\mu_0) \tag{10}$$

where $\phi(\mathbf{a}_i) = (a_{i,0}, -a_{i,N-1}, \ldots, -a_{i,1}) \in \mathbb{Z}_Q^N$ represents the coefficient vector derived from the polynomial mask, $\vec{s}'$ is derived from $\mathbf{S}$.

4) Key Switch. The extracted ciphertext has a dimension of kN and is encrypted under the key $\vec{s}'$. To revert to the original parameters $(n, \vec{s})$, a Key Switch operation is performed. We adopt the standard algorithm and parameterization described in [MP21]. Conceptually, it computes the linear sum of the public Key Switch Key based on the decomposed values of the input ciphertext.

5) Modulus Switch. The ciphertext modulus is scaled down from the bootstrapping modulus Q to the LWE modulus q to ensure parameter compatibility and correctness:

$$\mathbf{c}_{out} = \lfloor (q/Q) \cdot \mathbf{c}_{ks} \rceil \in \mathbb{Z}_q^{n+1} \tag{11}$$

Algorithm 1 TFHE Bootstrapping

Require: $\mathrm{LWE}_{\vec{s},n,q}(m+e)$ where $m = \frac{q}{p}m'$; TV
Ensure: $\mathrm{LWE}_{\vec{s},n,q}(\frac{q}{p}F(m') + \frac{q}{Q}e_{acc} + \frac{q}{q_{\mathrm{KS}}}(e_{ms} + e_{ks}) + e'_{ms})$

1: $\mathrm{GLWE}_{\mathbf{S},N,Q}(\mathrm{TV} \cdot X^{-(m+e)} + \mathbf{e}_{acc})$ ▷ (1) Blind Rotation
2: $\mathrm{LWE}_{\vec{s}',N,Q}(\frac{Q}{p}F(m') + e_{acc})$ ▷ (2) Sample Extract
3: $\mathrm{LWE}_{\vec{s}',N,q_{\mathrm{KS}}}(\frac{q_{\mathrm{KS}}}{p}F(m') + \frac{q_{\mathrm{KS}}}{Q}e_{acc} + e_{ms})$ ▷ (3) Modulus Switch to q_{KS}
4: $\mathrm{LWE}_{\vec{s},n,q_{\mathrm{KS}}}(\frac{q_{\mathrm{KS}}}{p}F(m') + \frac{q_{\mathrm{KS}}}{Q}e_{acc} + e_{ms} + e_{ks})$ ▷ (4) Key Switch
5: $\mathrm{LWE}_{\vec{s},n,q}(\frac{q}{p}F(m') + \frac{q}{Q}e_{acc} + \frac{q}{q_{\mathrm{KS}}}(e_{ms} + e_{ks}) + e'_{ms})$ ▷ (5) Modulus Switch to q
6: **return** $\mathrm{LWE}_{\vec{s},n,q}(\frac{q}{p}F(m') + \frac{q}{Q}e_{acc} + \frac{q}{q_{\mathrm{KS}}}(e_{ms} + e_{ks}) + e'_{ms})$

2.4 GPUs and On-Chip Memory

Modern GPUs utilize a Single Instruction Multiple Threads (SIMT) execution model to achieve massive parallelism. In this hierarchy, fundamental execution units called threads are grouped into warps (typically 32 threads) that execute instructions in lock-step. Multiple warps are further organized into Thread Blocks, which are scheduled onto Streaming Multiprocessors (SMs) where the warp scheduler performs zero-overhead context switching to hide arithmetic and memory latencies.

Table 3: On-Chip and Off-Chip Memory across Multiple Generations of GPUs

Specification	**NVIDIA A100**	**NVIDIA H100**	**NVIDIA RTX 4090**
Architecture	Ampere	Hopper	Ada Lovelace
Compute Capability	**8.0**	**9.0**	**8.9**
Streaming Multiprocessors (SMs)	108	132	128
On-Chip Memory (Per SM)			
Register File Size	256 KB	256 KB	256 KB
Max Shared Memory (SMEM)	**164 KB**	**228 KB**	**100 KB**
Constant Memory (CMEM)	64 KB	64 KB	64 KB
Off-Chip Memory			
Capacity	80 GB	80 GB	24 GB

Supporting this execution model is a stratified memory hierarchy, as detailed in Table 3. The architecture distinguishes between off-chip Global Memory (GMEM), which offers high capacity (e.g., 80 GB HBM3 on H100 [Cho23]) but incurs significant latency, and on-chip resources like registers and shared memory, which serve as a high-bandwidth, low-latency memory. Table 3 highlights the critical resource divergence between data center (A100, H100) and consumer (RTX 4090) platforms.

In this work, we target the NVIDIA RTX 4090. As illustrated in Table 3, this platform possesses significantly less shared memory compared to its data center counterparts. Nevertheless, we demonstrate that even under such constrained on-chip memory resources, our proposed BOLT-FHE scheme delivers substantial performance advantages.

3 Technical Overview

We provide a high-level overview of our proposed framework, starting with the mathematical formulation of the external product, followed by an analysis of the memory bottlenecks in state-of-the-art GPU implementations, and finally introducing our architecture-aware optimization strategy.

3.1 The Unified Process of GD- and MR-based GLWE $\times$ GGSW External Product.

The performance bottleneck of TFHE/FHEW bootstrapping lies in the blind rotation phase, which homomorphically evaluates the decryption circuit. Mathematically, this phase is composed of a sequence of external product operations between a bootstrapping key (encoded as GGSW ciphertexts) and a GLWE accumulator. While prior GPU-based acceleration efforts (e.g., [SYL+25, XLK+25]) have predominantly focused on the external product over RLWE and RGSW, we generalize this formulation to the GLWE and GGSW context to cover a broader range of parameter sets.

The selection of a noise management strategy fundamentally governs the underlying vector dimensionality and the requisite coefficient pre-processing pipeline. Specifically, existing approaches can be categorized as follows:

- **Gadget Decomposition (GD):** Mitigates noise growth by decomposing coefficients into small digits via a gadget matrix. This approach prioritizes noise stability but expands the ciphertext size proportional to the decomposition depth.
- **Modulus Raising (MR):** Conducts arithmetic under a large auxiliary modulus T and maps the result back to Q via scaling and rounding. MR maintains compact ciphertext sizes but necessitates higher precision for intermediate computations.

Figure 1 illustrates the divergent computation pipelines for GD- and MR-based external products. Abstractly, the external product between a GGSW ciphertext and a GLWE accumulator can be unified as a polynomial matrix-vector multiplication. To enhance computational efficiency, implementations typically migrate polynomial arithmetic to a transform domain[1] (e.g., NTT or FFT), converting computationally expensive convolutions into pointwise multiplications and accumulations.

3.2 Bootstrapping via On-chip Local Tiling (BOLT)

Translating the theoretical peak performance of GPUs into real-world FHE speedups presents significant architectural challenges. Unlike traditional compute-bound workloads, FHE operations are characterized by large ciphertext sizes, such as RLWE ciphertexts in BGV/CKKS or GLWE and GGSW ciphertexts in TFHE. Naively mapping these algorithms onto GPUs often results in severe memory bottlenecks. Consequently, existing GPU-based implementations [XLK+25, Zam24] often fail to fully saturate the available compute resources, highlighting the critical need for architecture-aware optimizations that prioritize data locality and on-chip resource utilization.

The aforementioned analysis in Section 1.1 reveals that the repeated materialization of a temporally-local working set makes blind rotation dominated by memory traffic, thereby preventing effective on-chip reuse of the ACC. To address this memory wall, we revisit bootstrapping from the perspective of GPU memory hierarchy and explicitly managed on-chip storage. Modern GPUs expose large register files and scratchpad memory (SMEM) with orders-of-magnitude higher bandwidth and lower latency than global memory.

[1] Since BOLT-FHE is based on integer instructions, all subsequent references to the transform domain refer to the NTT domain.

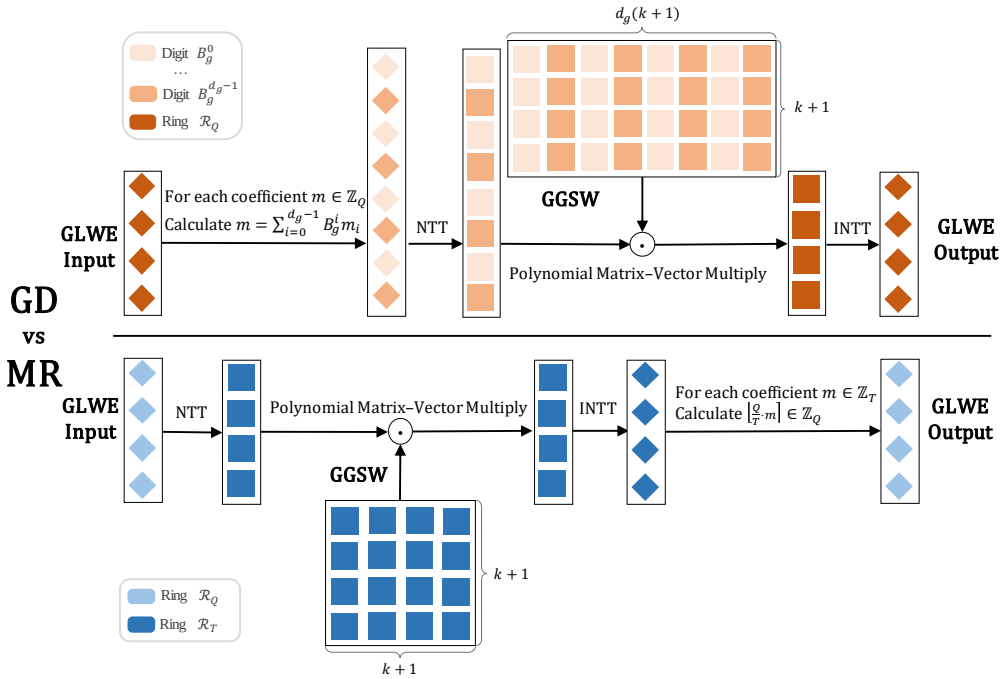

Figure 1: GLWE × GGSW External Product Computation Workflows under GD and MR. Example Configuration $k = 3$ and $d_g = 2$.

Our key observation is that the ACC working set of mainstream parameter regimes can fit within the on-chip budget, and thus the primary inefficiency of prior designs stems from organization, i.e., repeatedly breaking the ACC update chain by cross-kernel or cross-block interfaces. Driven by this insight, we propose BOLT-FHE, an aggressive on-chip local tiling strategy that constrains the critical accumulation path of external products to the lifetime of a single thread block and a single fused kernel whenever feasible. We map temporaries and partial sums to registers, use SMEM as the synchronization and staging buffer for NTT, and avoid repeated ACC materialization to global memory on the critical path. Accordingly, the dominant block-level SMEM footprint is the ACC itself, i.e., the $(k + r)$ accumulator polynomials preserved on-chip across both the (I)NTT transforms and the subsequent accumulation with GGSW. As a result, off-chip memory traffic is dominated by read-only key fetches and the final writeback rather than iterative store and load of the accumulator.

However, realizing such an on-chip local tiling strategy requires careful design to address the following challenges:

- **Scalability across diverse parameter sets.** Maintaining on-chip residency is nontrivial over the vast configuration space of generic GLWE × GGSW arithmetic. The on-chip footprint depends not only on the accumulator dimension $(k+1)$ and polynomial degree N, but also on the transient working sets induced by the chosen noise management path. In particular, under gadget decomposition (GD), naively materializing all d_g digits inflates the NTT workspace roughly proportional to $(k+1) \cdot d_g$, which can exceed SMEM budgets and forces partial spilling to off-chip memory. A robust design must therefore decouple the on-chip workspace from d_g and remain stable across parameter regimes, including larger k and higher security settings.

- **Throughput scaling under tight on-chip resource budgets.** To push through-

put beyond ciphertext efficiency, techniques such as CM packing [BBC+25, WLX+25] amortize blind rotation key usage by processing multiple accumulators under a shared mask. However, packing expands the effective vector/matrix dimension from $(k+1)$ to $(k+r)$ and increases the live working set within a block. Insufficient packing underutilizes on-chip resources, whereas overly aggressive packing amplifies register pressure and can trigger register spilling to local memory, severely degrading performance. Achieving stable throughput therefore requires systematic resource budgeting and design space exploration rather than ad-hoc tuning.

- **On-chip persistence across the arithmetic pipeline.** Blind rotation is not a single operator but a pipeline with frequent domain switches (e.g., coefficient domain ↔ NTT domain) and element-wise postprocessing. A modular multi-kernel organization introduces store and load boundaries that inherently break on-chip state and rematerialize intermediates in globally visible memory. Sustaining on-chip residency requires fusing the pipeline into a single execution template, while controlling register pressure and synchronization granularity so that fusion does not regress occupancy.

In what follows, we show that these challenges can be addressed by a fixed workspace external product formulation, a unified MegaKernel template with explicit resource knobs, and a systematic throughput scaling strategy via CM packing.

4 On-Chip External Product Computation

The computational workload during the ACC update phase of TFHE schemes is primarily dominated by the external product operations in the blind rotation. In this section, we systematically discuss how to compute external products under the constraints of limited GPU resources—specifically thread count, SMEM capacity, and register usage—while considering two noise reduction methods: Gadget Decomposition (GD) and Modulus Raising (MR).

4.1 GPU Parallelism and Design Trade-offs

Parallelism under a continuous external-product chain. In TFHE bootstrapping, blind rotation updates the ACC through a chain of external-product operations. A key constraint is that the update index along the LWE dimension, denoted by n, is inherently sequential: the i-th update consumes the ACC state produced by the $(i-1)$-th update, and thus cannot be parallelized without altering the algorithmic dependency structure. Consequently, practical GPU parallelism must come from (i) batching independent bootstrappings (throughput parallelism), and (ii) data-parallel execution within a single external product instance.

An external product is composed of (I)NTT, pointwise multiplications, and accumulations. A more intuitive view is to treat it as a dense polynomial matrix-vector multiplication, where computation is highly data-parallel over coefficients and polynomial blocks. However, the GPU design goal is not merely to accelerate an isolated matrix-vector multiply; instead, we require continuous execution of the update chain so that the ACC working set can be spatially reused across iterations. In this context, continuous explicitly emphasizes space reuse and state persistence: keeping the frequently updated ACC and its short-lived intermediates inside an on-chip workspace for as long as possible, rather than repeatedly breaking the chain by storing the ACC to GMEM between updates.

Why d_g-parallelism tends to induce global reduction pressure. A common strategy in prior GPU external product implementations is to expose parallelism along the gadget

decomposition dimension d_g by computing multiple digit contributions concurrently. This organization is reasonable for latency-oriented designs; however, in throughput-oriented settings it structurally requires a reduction step to merge per-digit partial results. Once such a reduction crosses kernel boundaries or thread-block boundaries, it inevitably falls back to GMEM, because under conventional GPU programming models the scope of SMEM is confined to a single thread block. As a result, d_g–parallelism often reintroduces globally visible materialization and merge traffic on the critical path, conflicting with the desired continuous on-chip reuse of ACC.

Motivated by the above trade-offs, we adopt a block-local execution model that keeps the blind rotation update chain continuous within a single thread block. We evaluate gadget decomposition in a digit-serial manner (serial over d_g) to avoid cross-block/kernel merges, while using intra-block parallel threads to partition the dense polynomial matrix–vector multiplication and (I)NTT.

4.2 Block-level On-chip Accumulator Workspace

To support the block-local continuous execution model in Section 4.1, we next specify how the external product is organized inside an explicit on-chip workspace. We first present a fixed workspace formulation for GD (with MR as the single-stage special case), and then show that the freed SMEM budget naturally enables CM packing for higher throughput.

Staged gadget decomposition for bounded SMEM. A key difficulty of keeping the external product update on-chip comes from the GD-induced expansion of temporaries. In GD, the accumulator is decomposed into d_g digits, and each digit must be transformed and multiplied with its corresponding key slice. If one materializes all digits (and their NTT domain) at once, the transient on-chip footprint grows roughly with $(k+1) \cdot d_g$, which can quickly exceed the SMEM budget and forces spills to off-chip memory. In contrast, MR does not introduce an explicit digit expansion in this sense, and thus naturally behaves like a single-stage update from the viewpoint of on-chip buffering.

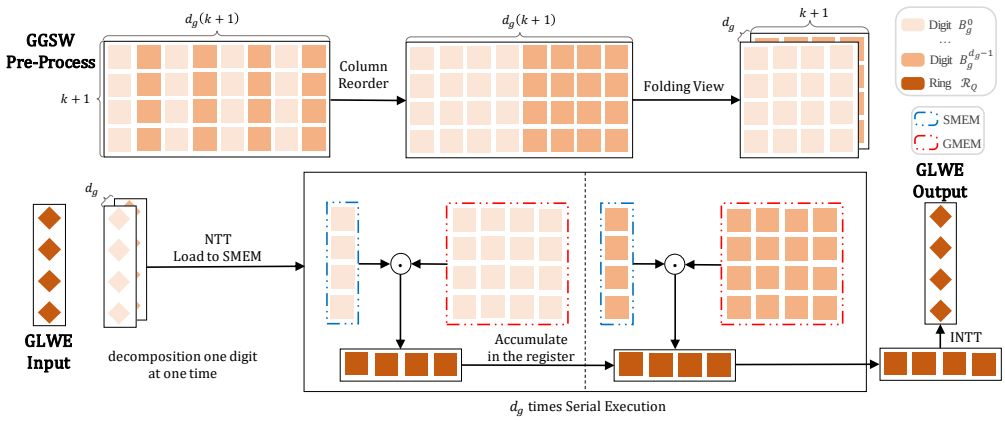

Figure 2: Staged External Product with Fixed SMEM Footprint. Example Configuration $k = 3$ and $d_g = 2$.

To remove the SMEM dependence on d_g, we evaluate GD digit by digit with a fixed-size SMEM buffer. As shown in Figure 2, we first adjust the bootstrapping key layout at key generation time. Instead of storing key slices grouped by polynomial first (the common layout where all digits of one polynomial are contiguous), we apply a simple column-wise reordering so that the same digit index across polynomials is stored contiguously. This reordering matches the staged execution order, enabling each stage ℓ to stream the needed

$\mathbf{A}^{(\ell)}$ slices with better locality and fewer strided accesses. At stage ℓ we only form the current digit $\mathbf{x}^{(\ell)}$, apply NTT, and immediately accumulate $\mathbf{A}^{(\ell)}\mathbf{x}^{(\ell)}$ into the running sum; after $\ell = 0, \ldots, d_g - 1$, we apply a single INTT and update ACC. This procedure is algebraically identical to the standard GD external product, $\mathbf{y} = \sum_{\ell=0}^{d_g-1} \mathbf{A}^{(\ell)}\mathbf{x}^{(\ell)}$, but it keeps the live on-chip footprint essentially constant (independent of d_g).

Further throughput optimization via CM packing. Recent work aims to give FHEW/TFHE bootstrapping SIMD-like throughput: amortized approaches pack many LWEs into a single RLWE to share the cost [MS18, GPVL23, DMKMS24], while we choose common-mask (CM) packing [BBC+25, WLX+25]. A CM-GLWE ciphertext encrypting $\mathbf{m} = (m_1, \ldots, m_r) \in \mathcal{R}_Q^r$ under a secret key matrix $\mathbf{S} \in \mathcal{R}_Q^{r \times k}$ can be written as $\mathbf{c} = (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_Q^{k+r}$, where $\mathbf{a} \in \mathcal{R}_Q^k$ is the shared mask and $\mathbf{b} = (b_1, \ldots, b_r) \in \mathcal{R}_Q^r$ contains the bodies. CM packs r message bodies $(b_1, \ldots, b_r)$ under a shared mask $\mathbf{a}$, enabling blind rotation to amortize key-related costs across multiple slots. The CM-bootstrapping building blocks are deferred to the Appendix B.

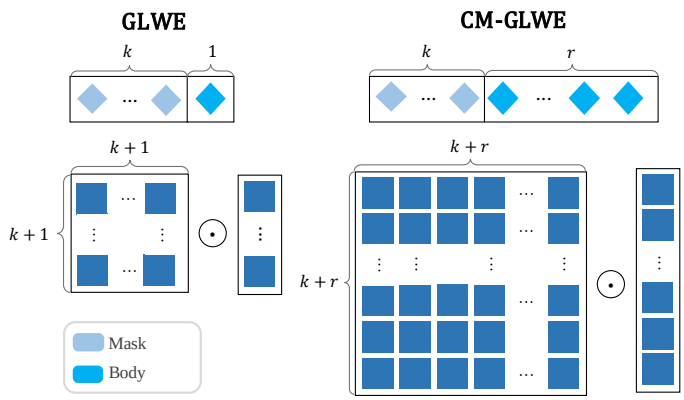

Figure 3: Expanding SMEM footprint with CM packing.

Our fixed workspace formulation provides a clean interface to incorporate CM without reintroducing the GD workspace blowup. CM expands the effective dimension from $(k+1)$ to $(k+r)$, while a naive GD implementation further multiplies the transient workspace by d_g, yielding an on-chip footprint on the order of $(k+r) \cdot d_g$ and making ACC residency difficult. With staged GD, however, GD contributes digit-serial accumulation rather than a d_g-fold space inflation, so the external-product update operates in a controlled $(k+r) \times (k+r)$ working mode. This makes GD/MR/CM combinations amenable to a single kernel template with predictable on-chip resource usage. Table 4 summarizes the resulting operator counts.

Table 4: Operator counts for blind rotation under GD/MR and CM packing. One (I)NTT counts either a forward NTT or an inverse NTT on a length-N polynomial.

Total #(I)NTTs			
Noise path	r Traditional Bootstrappings	CM Bootstrappings	Amortized CM
GD (ℓ=d_g)	$r\,n\,(k{+}1)(\ell{+}1)$	$n\,(k{+}r)(\ell{+}1)$	$n\left(\frac{k}{r}{+}1\right)(\ell{+}1)$
MR (ℓ=1)	$2r\,n\,(k{+}1)$	$2n\,(k{+}r)$	$2n\left(\frac{k}{r}{+}1\right)$
Total # pointwise multiplications			
Noise path	r Traditional Bootstrappings	CM Bootstrappings	Amortized CM
GD (ℓ=d_g)	$r\,n\,(k{+}1)^2\,\ell\,N$	$n\,(k{+}r)^2\,\ell\,N$	$n\,\frac{(k+r)^2}{r}\,\ell\,N$
MR (ℓ=1)	$r\,n\,(k{+}1)^2\,N$	$n\,(k{+}r)^2\,N$	$n\,\frac{(k+r)^2}{r}\,N$

n: LWE dimension; k: GLWE mask dimension; N: polynomial degree; r: CM packing factor (number of packed bodies); ℓ: decomposition levels (GD: $\ell = d_g$, MR: $\ell = 1$).

As r increases, the number of (I)NTTs per packed message is amortized down (the k/r term in Table 4), while the pointwise multiplications grow with the expanded dimension. Consequently, the amortized pointwise multiplies term $\frac{(k+r)^2}{r}$ exhibits a minimum at an intermediate r, implying an optimal packing factor under practical compute/memory budgets.

5 Unified Blind Rotation Kernel Design

In this section, we present a unified blind rotation kernel design and implementation that supports gadget decomposition (GD), modulus raising (MR), and common mask (CM) packing under varying rank k of GLWE within the same structure. Blind rotation is a sequence of external products, whose inner loop is dominated by dense polynomial matrix–vector multiplies and repeated NTT/INTT domain switches. Accordingly, our kernel is organized at warp granularity and uses two compile-time parameters to control work partition and register footprint:

WPP and IPT. WPP (warps per polynomial) specifies how many warps cooperatively process one polynomial in the in-place NTT/INTT and in the pointwise multiply and accumulate stages. IPT (items per thread) specifies how many coefficients are processed by each thread for that polynomial. Given the fixed polynomial degree N, we set

$$32 \cdot \mathrm{WPP} \cdot \mathrm{IPT} = N, \tag{12}$$

so each warp thread is responsible for a uniform slice of coefficients. WPP mainly controls the degree of multi-warp cooperation (and thus the number of passes when scanning sub-NTTs), while IPT primarily budgets the per-thread register usage for coefficient tiles and NTT-domain accumulators. These two knobs are used consistently in the NTT design (Section 5.1) and in the fused MegaKernel (Section 5.2).

5.1 Packed Radix-2 Sub-Warp NTT

In TFHE settings, the polynomial degree is typically chosen from $N \in \{512, 1024, 2048\}$. To integrate NTT/INTT tightly into the blind rotation MegaKernel, we adopt a 4-step decomposition that rewrites an N-point transform into a collection of small, regular sub-NTTs (e.g., 16-, 32-, and 64-point). Specifically, the 4-step method views the N coefficients as a matrix, performing sub-NTTs on columns and rows interleaved with pointwise twiddle factor multiplications and matrix transpositions. A fixed number of warps then scan these

sub-tasks to complete the full transform. Each sub-NTT uses radix-2 butterflies (Cooley–Tukey [CT65]/Gentleman–Sande [GS66]), which keeps the implementation uniform across different N and simplifies maintenance.

To match shared memory reuse and the column/row access pattern induced by the 4-step layout, our blind rotation kernel employs a packed sub-warp execution scheme: lanes within a warp are partitioned into groups so that a single warp processes multiple sub-NTTs in parallel (e.g., two 32-point sub-NTTs, or four 16-point sub-NTTs). The key observation is that the 32 lanes of a warp can cover up to 32 butterfly pairs per stage, allowing us to fix the parallel granularity at the warp level and confine data exchange to shared memory.

Lane-group parallelism within a warp. We derive a group identifier from the high bits of the lane index to select the sub-NTT instance (and its row/column position), and use the low bits to determine the butterfly-pair role within that sub-NTT. Each stage follows a fixed "load–butterfly–store" pattern and performs in-place updates in shared memory. The `__syncwarp()` barrier is inserted between consecutive stages to enforce the read-after-write dependency, ensuring that writes from all lanes are visible before the next stage proceeds and preventing races caused by lane progress skew.

Multi-warp cooperation via WPP. We parameterize the work partition using WPP (warps per polynomial). For the 1024-point case in the column-transform phase, the 4-step decomposition yields 32 independent 32-point sub-NTTs. With WPP= 1, a single warp must scan all 32 sub-tasks, requiring 16 serialized passes under the packed scheme. With WPP= 4, these sub-tasks are distributed across 16 warps, reducing the scan to 4 passes. The transition between the column and row phases requires one block-level barrier `__syncthreads()` to ensure shared memory consistency across warps.

Operator fusion and shared memory layout. We fuse the pointwise multiplications by the twiddle factors into the last (or first, depending on NTT versus INTT) butterfly stage, so that butterfly outputs are multiplied and written back immediately. This avoids an additional shared memory pass. In addition, the shared memory layout uses padding to mitigate bank conflicts in the column/row access pattern, stabilizing bandwidth utilization during the in-place stages.

5.2 The MegaKernel: Fusing Computation with Optimized Memory Access

To maximize instruction throughput and minimize global memory traffic, we consolidate the initialization, blind rotation, sample extraction, and modulus switching phases into a single fused kernel, which we term the MegaKernel. We note that the Key Switch operation is implemented as a standalone kernel due to its distinct thread organization requirements, following the approach in [SYL+25]. Given that Key Switch accounts for a negligible fraction of the total execution time ($< 10\%$), we omit its detailed discussion to focus on the bottleneck operations handled by the MegaKernel.

Prior GPU bootstrapping implementations [SYL+25, XLK+25] are typically tailored to the RLWE setting ($k = 1$). In contrast, our implemented MegaKernel is parameterized for general GLWE ciphertexts with arbitrary k, thereby directly supporting advanced parameter sets (WPP and IPT) where the accumulator consists of multiple polynomial components.

Another key enabler of versatility is the employment of staged digit processing, as introduced in Section 4.2. This technique decouples the on-chip memory footprint from the total digit count, enabling the same kernel structure to efficiently handle a wide range

of gadget decomposition depths d_g without requiring a redesign of the memory layout or execution flow. Consequently, the kernel admits both gadget decomposition and modulus raising as selectable configurations within the same execution template.

Figure 4 depicts the execution template of our blind rotation MegaKernel within a single thread block. The block is organized as `dim3`$(32(k+r), \mathrm{WPP}, 1)$, where WPP warps are assigned to handle each of the $(k+r)$ polynomials. Through this flexible parameterization, our design ensures sufficient parallelism across diverse parameter sets while maintaining optimal data locality.

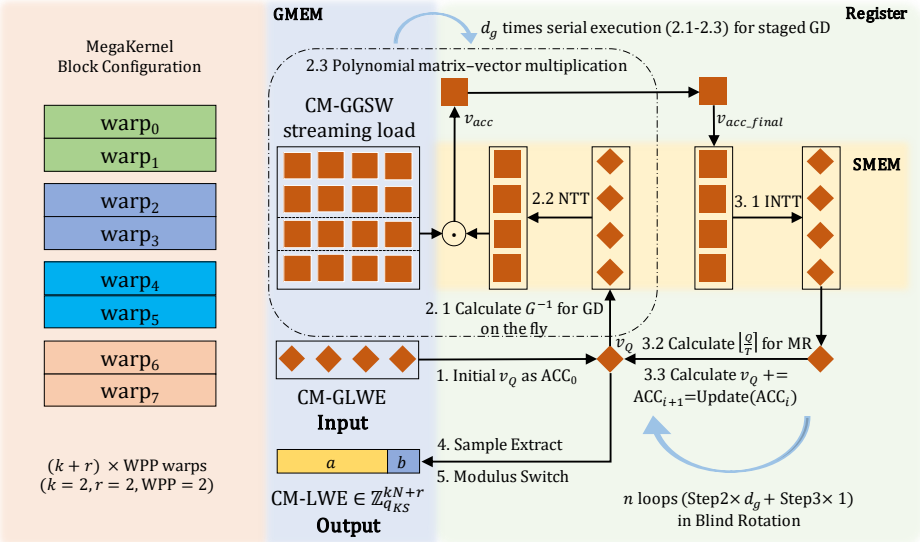

Figure 4: Overview of the Blind Rotation MegaKernel with Example Configuration WPP = 2, $k = 2$, and $r = 2$.

The MegaKernel executes as follows:

Step 1: Initialization and Tiling. At entry, the ACC is loaded from GMEM into per-thread registers. Data is organized as coefficient tiles, ensuring that the initial state resides entirely in the fastest on-chip memory before computation begins.

Step 2: Staged Gadget Decomposition and Multiplication. This phase executes the multiplication and addition of the decomposed ACC digits and the CM-GGSW directly in NTT form. Following the staged schedule from Section 4.2, the loop processes gadget digits serially to manage SMEM pressure:

2.1 Serial Staging GD: For Gadget Decomposition, only one digit slice is staged into SMEM per stage on the fly. This minimizes the SMEM footprint required for the subsequent transform.

2.2 NTT Transformation: The staged digit slice in SMEM is transformed to the NTT domain.

2.3 Multiply and Add: The streamed CM-GGSW slices are multiplied with the NTT-form digit and accumulated into partial sums.

Step 3: INTT and Fused Update. Once the inner multiplication loop completes, the accumulator is updated. Both GD and MR phases share the same transform infrastructure but differ at the staging boundaries:

3.1 INTT: The partial sums are written to SMEM to perform an in-place INTT.

3.2 Fused Modulus Raising: Unlike a standard modular reduction, we fuse the scale-and-round step directly when reading the INTT results back from SMEM. Consequently, the coefficient-domain accumulator tiles in registers are updated directly into their target representation for the next iteration, without extra memory round-trips.

Steps 4 & 5: Output. After completing all n steps of the blind rotation, the kernel executes an element-wise fusion of Sample Extraction and Modulus Switching. The final CM-LWE result is then directly written to GMEM, completing the pipeline.

Following the MegaKernel, the CM-LWE ciphertext is processed by a Key Switch kernel. As a typical memory-bound kernel, it performs a streaming load of the KSK to execute modular additions with the ciphertext. We similarly fuse a modulus switching operation at the output stage to complete the entire bootstrapping process.

Potential for hybrid noise control methods. As shown in Figure 4, our MegaKernel supports not only distinct GD and MR modes but also hybrid noise-control compositions with only lightweight changes. The underlying execution flow requires no modification, as the operand preparation steps can be combined flexibly. Since hybrid schemes are currently less common, we defer the specific parameter optimization for hybrid noise reduction to future investigations.

Implementation details. We represent NTT-domain operands in Montgomery form and implement modular products via Montgomery multiplication [Mon85] with lightweight reductions along the multiply and add path. Bootstrapping-key (in CM-GGSW ciphertext) slices are read from GMEM using coalesced accesses. The accumulator workspace is accessed from SMEM using a padded, strided layout to mitigate bank conflicts.

5.3 SMEM and Register Budgeting of the Fused MegaKernel

The fused MegaKernel is constrained jointly by SMEM and registers. For a feasible launch configuration, the block shape, SMEM footprint, and per-thread register usage must be satisfied simultaneously.

We first consider SMEM. In the packed setting, each thread block maintains $(k+r)$ ACC polynomials on chip. In addition, we cache in SMEM the precomputed table used by the four-step (I)NTT, whose size is approximately that of two more N-coefficient polynomials. Therefore, ignoring minor padding and alignment overheads, the dominant block-level SMEM budget is

$$\mathrm{SMEM}_{\mathrm{block}} \approx (k+r+2) \cdot N \cdot \texttt{sizeof(uint64)}. \tag{13}$$

This budget is coupled directly to the block organization $\texttt{dim3}(32(k+r), \mathrm{WPP}, 1)$. Increasing r enlarges the ACC workspace, while increasing WPP increases the number of warps per block. Thus, r, N, and WPP must be chosen jointly to satisfy the SMEM, block size, and residency constraints. Due to staged digit processing, this dominant SMEM footprint does not scale with d_g.

Register budgeting is less direct, since register usage is assigned by the compiler rather than specified explicitly at the source level. This issue is amplified in the fused MegaKernel, which spans multiple arithmetic stages and contains nested device functions, including the four-step (I)NTT and the accumulation of GGSW key slices. Under aggressive optimization settings, excessive inlining and loop unrolling can enlarge live ranges and inflate register usage beyond a feasible budget.

To control this effect, we apply register budgeting with explicit compiler considerations. At the function level, we use `forceinline` only for small device functions, while marking larger device functions as `noinline` to prevent excessive live range growth across long

Table 5: Per-block SMEM budget of the fused MegaKernel and representative instantiations under the RTX 4090 block-level SMEM limit.

Dominant per-block SMEM budget			
Component	**Footprint per block**		
ACC workspace	$(k+r) \cdot N \cdot$ `sizeof(uint64)`		
Four-step (I)NTT support table	$2 \cdot N \cdot$ `sizeof(uint64)`		
Total dominant SMEM	$(k+r+2) \cdot N \cdot$ `sizeof(uint64)`		
Instantiations under 100 KB max SMEM (RTX 4090)			
N	**One polynomial**	**SMEM / block**	**Approx. max** $(k+r)$
512	4 KB	$(k+r+2) \cdot 4$ KB	$k+r \lesssim 22$
1024	8 KB	$(k+r+2) \cdot 8$ KB	$k+r \lesssim 10$
2048	16 KB	$(k+r+2) \cdot 16$ KB	$k+r \lesssim 4$

arithmetic sequences. At the loop level, we treat IPT loops selectively: simple loops, such as those used for the accumulation of GGSW key slices, can be fully unrolled, whereas the loops inside the NTT device functions are explicitly prevented from aggressive unrolling, since full unrolling there tends to inflate the number of simultaneously live temporaries and significantly increase register usage. At the kernel level, we annotate the fused blind rotation kernel with `launch_bounds` to guide the compiler toward a feasible register budget under the intended residency. Overall, the final implementation intentionally trades some compile time ILP for a more feasible fused execution. In practice, this design keeps register pressure manageable for the main operating points, although the actual spill behavior still depends on the specific WPP/IPT configuration.

6 Evaluation

Experimental Setup. Unless otherwise specified, all experiments are conducted on an NVIDIA RTX 4090 GPU and an Intel Xeon Silver 4410Y CPU @2.0 GHz with 128 GB RAM. The software stack is Ubuntu 24.04 with `gcc` 13.3 and full CUDA 12.9.1 toolkit. For the datacenter GPU evaluation, profiling results are collected on an NVIDIA A100-40GB, while the reported throughput is measured on an NVIDIA A800-80GB. All measurements use a batch size of 16,384 bootstrapping instances. For CM packing, if one thread block processes r packed ciphertexts, we set the batch size to $16{,}384/r$ so that the total number of bootstrapping instances remains matched across different r. All reported performance metrics represent end-to-end execution times, encompassing the total latency incurred by ciphertext transfers between the host CPU and the GPU device.

Open-source implementation. The source code of BOLT-FHE is publicly available at https://github.com/Leafxiaotong/BOLT-FHE.

6.1 Parameter Configuration

We have conducted an extensive benchmarking evaluation against a broad range of widely-adopted parameter sets. To facilitate a fair comparison with existing SOTA GPU implementations, the provenance of all evaluated parameters is explicitly detailed in Table 6. For parameter sets G1–G3 and M1–M6, evaluations are conducted using homomorphic NAND gates. For G4–G5, we employ the TFHE instantiation within the OpenFHE [BAB+22] library. All experiments utilize a uniform ternary secret key distribution.

Specifically, parameters associated with the GD framework are sourced entirely from the OpenFHE library. Regarding the MR framework, parameter sets M1–M2 are derived

Table 6: Parameter Sets

ParamID	Source	λ	n	q	N	k	B_g	$\log_2 Q$	$\log_2 T$	Q_{KS}	B_{KS}
Gadget Decomposition (GD)											
G1	OpenFHE STD128	128	503	1024	1024	1	2^8	27	–	2^{14}	2^5
G2	OpenFHE STD128Q3	128	600	2048	2048	1	2^{25}	50	–	2^{15}	2^5
G3	OpenFHE EvalBinGate	128	512	1024	1024	1	2^7	27	–	2^{14}	2^5
G4	OpenFHE EvalFunc	128	1305	2048	2048	1	2^{27}	54	–	2^{35}	2^5
G5	OpenFHE EvalFloor	128	1305	2048	1024	1	2^5	27	–	2^{35}	2^5
Modulus Raising (MR)											
M1	[LLL+24] T_1024_36	96	512	512	1024	1	–	16	36	2^{14}	2^9
M2	[LLL+24] T_2048_50	128	1024	1024	2048	1	–	22	50	2^{21}	2^5
M3	[WLX+25] STD128	128	512	1024	512	3	–	18	41	2^{14}	2^7
M4	[WLX+25] STD128-I	128	512	2048	1024	2	–	18	41	2^{14}	2^7
M5	[WLX+25] STD256	256	1088	2048	1024	3	–	19	44	2^{15}	2^5
M6	[WLX+25] STD192	192	1024	1024	2048	1	–	17	37	2^{16}	2^8

from [LLL+24], while ParamID M3–M6 are adopted from [WLX+25]. It is noteworthy that while the original MR configurations in [LLL+24] and VeloFHE [SYL+25] utilize composite moduli (e.g., PQ) and the corresponding composite-modulus NTT, we have unified these into the framework proposed by [WLX+25]. Our approach maintains parameter consistency while setting the ciphertext modulus Q as a power of two. We introduce an auxiliary modulus T to represent the expanded computational domain, where T is selected as an NTT-friendly prime of a magnitude comparable to the original composite PQ product in [LLL+24].

Furthermore, observing that several MR parameter sets in the literature do not explicitly account for the impact of the packing factor r on the decryption failure probability, we have re-evaluated these probabilities using the noise modeling framework introduced in [WLX+25] to ensure the rationality (see Appendix A).

Table 7: Decryption Failure Probability Bounds ($-\log_2 p_{\mathrm{fail}}$) for MR Parameter Sets M1–M6

	ParamID: M1			**ParamID: M2**			**ParamID: M3**		
r	σ_{tot}	$-\log_2 p_{\mathrm{fail}}$	Δbits	σ_{tot}	$-\log_2 p_{\mathrm{fail}}$	Δbits	σ_{tot}	$-\log_2 p_{\mathrm{fail}}$	Δbits
1	18.2935	26.74†	0.00	18.9254	102.66	0.00	30.8836	37.93	0.00
2	18.5107	26.09	−0.65	18.9255	102.66	≈ 0	30.8876	37.92	−0.01
4	18.9375	24.88	−1.85	18.9256	102.66	≈ 0	30.8957	37.90	−0.03
8	19.7636	22.76	−3.97	18.9258	102.66	≈ 0	30.9119	37.86	−0.07
16	21.3199	19.42	−7.32	18.9262	102.65	−0.01	30.9442	37.78	−0.15

	ParamID: M4			**ParamID: M5**			**ParamID: M6**		
r	σ_{tot}	$-\log_2 p_{\mathrm{fail}}$	Δbits	σ_{tot}	$-\log_2 p_{\mathrm{fail}}$	Δbits	σ_{tot}	$-\log_2 p_{\mathrm{fail}}$	Δbits
1	65.6569	33.45	0.00	52.0111	53.90	0.00	26.8835	50.37	0.00
2	65.6721	33.44	−0.02	52.0136	53.90	−0.01	29.1650	42.65	−7.72
4	65.7026	33.40	−0.05	52.0187	53.89	−0.02	33.2619	32.56	−17.81
16	65.8848	33.21	−0.24	52.0493	53.82	−0.08	51.3894	13.06	−37.31

† [WLX+25] adopts a more conservative model than [LLL+24], resulting in higher p_{fail} estimates (T_1024_36 $\approx 2^{-40}$ in [LLL+24]).

Table 7 summarizes the decryption failure probability bounds ($-\log_2 p_{\mathrm{fail}}$) and the corresponding differences (denoted as Δbits) under various packing factors r. It is important to note that the noise modeling framework adopted from [WLX+25] differs from prior works [MP21]; their approach is more conservative, leading to looser noise upper bounds. For instance, for the same parameter set M1, the noise model in [LLL+24] yields a failure probability of 2^{-40}, whereas the model in [WLX+25] results in 2^{-26}.

For parameter sets with a large scale factor T/Q^2 (e.g., M2–M5), the increment in p_{fail}

remains marginal even as r increases to 16. In contrast, for parameter sets with tighter noise budgets, such as M1 and M6, the failure probability increases significantly, suggesting they are less suitable for high packing factors. We clarify that Table 7 is provided for illustrative sensitivity analysis, and we do not employ impractical packing factors for M1 and M6 in our actual benchmarks.

6.2 Performance Evaluation and Comparison

Since the choice of configuration determines the packing factor r, we first demonstrate the tuning results for M1–M6. For G1–G5, a comparison with SOTA implementations [SYL+25, XLK+25] is provided in the end.

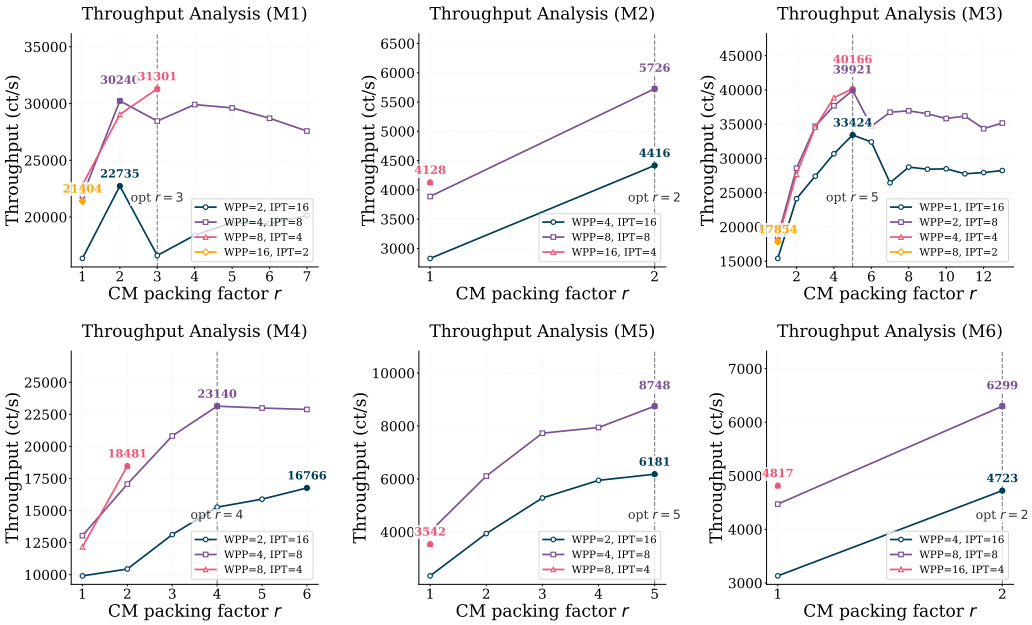

Figure 5: Fine-tuning Results of Template Parameters WPP and IPT for ParamID M1–M6.

Performance optimization via WPP and IPT tuning. We conduct extensive performance evaluations across six parameter sets, denoted M1–M6. By leveraging our kernel template parameters, WPP and IPT, we are able to fine-tune the thread-block organization and the register pressure per thread. Notably, WPP governs the maximum number of polynomials residing in a single thread block, which is primarily constrained by the available SMEM capacity. The sum of the module rank k and the packing factor r is bounded by the interplay of these hardware resources and WPP. Since k remains constant for a target security level, we optimize the system throughput by adjusting the packing factor r. Figure 5 summarizes the tuning result. Table 8 details the thread-block organization for each parameter set, along with the corresponding range of the packing factor r evaluated in our design space exploration.

As illustrated in Figure 5, for the ParamID M3, under the configuration with WPP= 8 and IPT= 2, the packing factor r is restricted to 1, leading to the lowest observed throughput. Conversely, the other three configurations achieve their peak throughput at $r = 5$. This trend results from the multi-dimensional resource constraints within a single GPU block. For ParamID M3, we have $N = 512$ and $k = 3$. With WPP= 4

and $r = 5$, the block maintains $(k + r) = 8$ ACC polynomials, so the ACC occupies approximately $8 \times 512 \times 8 \approx 32$ KB of shared memory. Adding the NTT precomputation table (approximately 8 KB), the total shared memory usage becomes approximately 40 KB per block. In this case, selecting WPP= 2 or WPP= 4 provides the most favorable block residency (with 2 or 1 blocks residing on each SM, respectively), which explains why these configurations achieve the highest throughput. These results further validate the efficiency and versatility of our architectural design.

Table 8: Exploration of Kernel Template Parameters and Packing Factor r. Each cell denotes: (i) (WPP, IPT) organization, (ii) the allowed r range constrained by max SMEM configuration on RTX 4090, and (iii) which r achieves highest throughput (r, Throughput [ct/s]).

ParamID	(N, k)	**Thread-block Organization (WPP, IPT)**			
		I	**II**	**III**	**IV**
M1	$(1024, 1)$	(2, 16) | $r \in [1, 7]$ (2, 22,735)	(4, 8) | $r \in [1, 7]$ (2, 30,240)	(8, 4) | $r \in [1, 3]$ **(3, 31,301)***	(16, 2) | $r \in [1, 1]$ (1, 21,404)
M2	$(2048, 1)$	(4, 16) | $r \in [1, 2]$ (2, 4,416)	(8, 8) | $r \in [1, 2]$ **(2, 5,726)***	(16, 4) | $r \in [1, 1]$ (1, 4,128)	—
M3	$(512, 3)$	(1, 16) | $r \in [1, 13]$ (5, 33,424)	(2, 8) | $r \in [1, 13]$ (5, 39,921)	(4, 4) | $r \in [1, 5]$ **(5, 40,166)***	(8, 2) | $r \in [1, 1]$ (1, 17,854)
M4	$(1024, 2)$	(2, 16) | $r \in [1, 6]$ (6, 16,766)	(4, 8) | $r \in [1, 6]$ **(4, 23,140)***	(8, 4) | $r \in [1, 2]$ (2, 18,481)	—
M5	$(1024, 3)$	(2, 16) | $r \in [1, 5]$ (5, 6,181)	(4, 8) | $r \in [1, 5]$ **(5, 8,748)***	(8, 4) | $r \in [1, 1]$ (1, 3,542)	—
M6	$(2048, 1)$	(4, 16) | $r \in [1, 2]$ (2, 4,723)	(8, 8) | $r \in [1, 2]$ **(2, 6,299)***	(16, 4) | $r \in [1, 1]$ (1, 4,817)	—

$\star$ Denotes the optimal design choice for the specific parameter set.

It is important to note that M4 and M5 utilize MLWE structures. The adoption of MLWE instead of RLWE is driven by the MR technique, which necessitates an exceptionally large modulus T to maintain security strength. Our implementation demonstrates that module lattices are highly compatible with GPU architectures, specifically through our fixed workspace design.

Optimized performance and comparison with CPU baseline. Table 9 presents a comprehensive performance comparison between our proposed GPU implementation and single-threaded CPU baselines across six distinct ParamID M1–M6.

It is important to note that, to the best of our knowledge, there are currently no existing hardware accelerators that optimize GLWE and GGSW ciphertexts specifically using the CM packing strategy. Due to the absence of comparable GPU implementations for this specific algorithmic configuration, we established our baseline by directly adopting the CPU performance results reported in Table 9 of [LLL+24] and Table 2 of [WLX+25]. Against the baselines, our solution running on the NVIDIA RTX 4090 achieves a substantial speedup, consistently outperforming the CPU execution by approximately three orders of magnitude. Specifically, the speedup factors range from 786× (M2) to a peak of 1592× (M5), effectively bridging the gap between theoretical lattice cryptography and practical deployment.

Architectural profiling. To further explain the observed tuning and throughput behavior, we next provide a microarchitectural analysis based on profiling results. As shown in

Table 9: Performance Comparison on RTX 4090 vs CPU Baselines.

ParamID	Opt. r	FP	Ours (RTX 4090) Thr. (ct/s)	CPU Baseline Lat. (ms)†	Speedup
M1	3	$2^{-25.47}$	31,301	34.38	**1076.13**×
M2	2	$2^{-102.66}$	5,726	137.44	**786.97**×
M3	5	$2^{-37.89}$	40,166	25.80	**1036.28**×
M4	4	$2^{-33.40}$	23,140	57.00	**1318.96**×
M5	5	$2^{-53.88}$	8,748	182.00	**1592.08**×
M6	2	$2^{-42.65}$	6,299	183.00	**1152.66**×

†: CPU baseline latency for M1–M6 is retrieved from single-threaded results in Table 9 of [LLL+24] and Table 2 of [WLX+25].

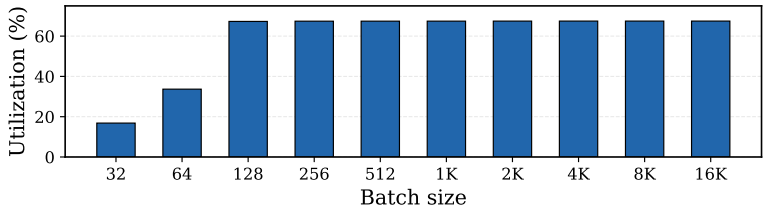

Figure 6: SM utilization under different batch sizes for ParamID G2 on RTX 4090.

Figure 6, SM utilization rises quickly with batch size and saturates at batch size 128. By comparison, the achieved occupancy remains almost constant at about 66.7% over the whole range, and the L1/TEX throughput also stays nearly constant at about 50.9%. Therefore, the main effect of increasing batch size is to improve effective SM utilization, rather than changing occupancy or on-chip memory throughput.

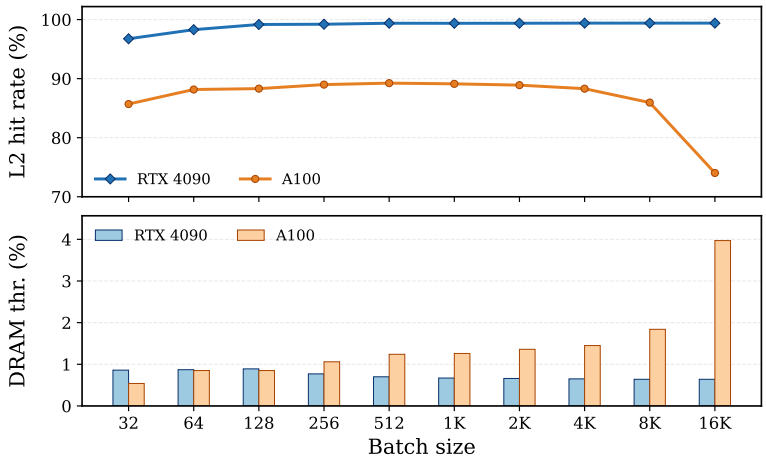

Figure 7: L2 cache behavior under different batch sizes for ParamID G2.

Table 10 reports the register allocation and spill results for the ParamID M1 under different IPT/WPP settings. As IPT increases, each thread holds more polynomial fragments, thereby increasing register pressure and leading to larger stack frames and more spill stores/loads. Notably, the best throughput does not occur at the spill-free point. Instead, the highest throughput is achieved at $(\mathsf{IPT}, \mathsf{WPP}) = (4, 8)$ with $r = 3$, showing that the optimal setting is determined by the trade-off between packing efficiency and register

Table 10: Register Allocation and Spill Count for ParamID M1 on RTX 4090.

IPT	WPP	Reg/thread† (regs)	Stack frame† (Bytes)	Spill stores† (Bytes)	Spill loads† (Bytes)	Opt. r (Max. r)‡	Peak throughput (ct/s)
2	16	64	0	0	0	1 (1)	21,403
4	8	64	24	32	32	3 (3)	31,301
8	4	64	32	64	80	2 (7)	30,239
16	2	64	320	564	688	2 (15)	22,735

† Reg/thread, stack frame, spill stores, and spill loads are reported by `ptxas`.
‡ Under CM packing for ParamID M1 ($k = 1$), Max. r is computed by considering only the warp constraint under the current WPP setting (with at most 32 warps per block), without taking the shared memory constraint.

pressure rather than by spill minimization alone. In practice, the dominant working set remains on-chip, and only a small number of temporaries may spill under more aggressive settings.

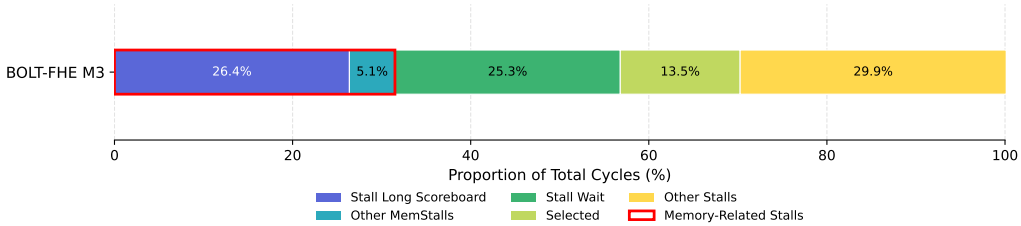

Figure 8: Warp state breakdown of the fused kernel for ParamID M3.

We further report the warp state breakdown for ParamID M3 configuration in Figure 8. We group the reported warp stalls into Memory-Related stalls, Stall Wait, Selected, and other stalls. As shown, Memory-Related stalls constitute a major fraction of the total cycles, while Stall Wait is also a prominent component. This indicates that, for M3 configuration, the fused MegaKernel is mainly limited by long latency dependencies rather than severe pipeline throttling effects.

Comparison with prior GPU-based works. We evaluate the concrete performance of our implementation in comparison with SOTA GPU implementations [SYL+25, XLK+25] for TFHE. To this end, we report performance metrics for the GD method and the MR method as utilized in VeloFHE [SYL+25].

As shown in Table 11 and Table 12, our implementation achieves a significant speedup across the majority of the evaluated parameter sets. The parallelization focus of [XLK+25] is on enhancing the parameter scalability for individual ciphertexts, pursuing extreme parallelism to a certain extent. In contrast, our implementation emphasizes throughput efficiency in large batch gate-level parallelism, improving the effective computational density per gate by minimizing cross-block/kernel synchronization and intermediate state exchanges. Similarly, VeloFHE [SYL+25] also focuses on improving bootstrapping efficiency by reducing the global round-trips of intermediate states. However, the specific organization of the blind rotation/external product chain within the kernel is not fully disclosed in the public text. To ensure reproducibility and comparability, we only use the end-to-end performance data reported in their paper.

For the datacenter platform results in Table 11, the advantage of BOLT-FHE becomes smaller than on RTX 4090, which is consistent with an explanation based on cache capacity. In particular, RTX 4090 provides a 72 MB L2 cache, whereas A100 provides a 40 MB L2

Table 11: Performance comparison with VeloFHE [SYL+25].

ParamID	Ours	VeloFHE [SYL+25]	Speedup
	Thr. (ct/s)	Thr. (ct/s)	
G1	31,878 (16,744)	11,378 (7,190)	**2.80**× (2.33×)
G2	9,489 (4,453)	3,249 (2,686)	**2.92**× (1.66×)
M1	23,442 (12,129)	9,871 (5,969)	**2.38**× (2.03×)
	31,301 (15,417)‡	–	**3.17**× (2.58×)
M2	4,187 (2,683)	1,729 (1,578)	**2.42**× (1.70×)
	5,726 (3,580)‡	–	**3.31**× (2.27×)

‡: Optimal performance with CM packing. For M1, $r = 3$ on both RTX 4090 and A800-80GB; for M2, $r = 2$ on RTX 4090 and $r = 3$ on A800-80GB.
For [SYL+25], data is based on TFHE gate bootstrapping results.
G1 uses the updated `uint32` implementation.

Table 12: Performance comparison on G3, G4, and G5 on RTX 4090.

ParamID	Ours	VeloFHE [SYL+25]		[XLK+25]	
	Thr. (ct/s)	Thr. (ct/s)	Speedup	Thr. (ct/s)	Speedup
G3	30,813	11,111†	**2.77**×	4,348†	**7.09**×
G4	2,660	1,408†	**1.89**×	629†	**4.23**×
G5	2,576	1,389†	**1.85**×	595†	**4.33**×

†: Calculated from reported latency as 1000/latency(ms).
Data for [SYL+25] corresponds to the "VeloFHE-T" ternary key variant.
Data for [XLK+25] is based on their RTX 4090 homomorphic function benchmarks.
G3, G5 use the updated `uint32` implementation, G5 reports complete EvalFloor throughput (two PBS calls per input).

cache. Since BOLT-FHE relies on a compact working set and benefits more when hot data can remain resident in L2, a smaller L2 cache weakens cache coverage and causes more accesses to fall back to DRAM. This interpretation is consistent with the profiling results in Figure 7, where A100 shows a lower L2 hit rate and a higher DRAM throughput than RTX 4090. Overall, these results suggest that BOLT-FHE is especially favorable on GPU architectures with larger L2 caches, such as newer GPUs including the RTX 5090 and the Blackwell B200, which provide 96 MB and 126 MB of L2 cache, respectively.

7 Conclusion

This paper introduces BOLT-FHE, a unified acceleration framework designed to align general FHE programming paradigms with the massive parallelism and memory hierarchy of modern GPUs. By identifying the memory bottlenecks in prior TFHE implementations, we developed an aggressive on-chip residency strategy that meticulously maps the entire blind rotation accumulation to registers and shared memory, effectively eliminating excessive DRAM traffic. Furthermore, we leverage surplus on-chip capacity through Common-Mask packing and a fused Megakernel design, which significantly amplifies throughput by enabling concurrent execution. Future work will explore more parameter configurations and GPU's latest hardware features (e.g., Tensor Core [MDCL+18] and DSM [Cho23]) for FHE.

Acknowledgments

We would like to thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions. We are grateful to the anonymous shepherd for helping us to improve our paper. This work is supported in part by the National Cryptographic Science Foundation of China under Grant No. 2025NCSF02005, National Natural Science Foundation of China under Grant No. 62572255, 62302238, 61902392; in part by the China Postdoctoral Science Foundation under Grant No. 2025T180411; in part by CCF-Huawei Populus Grove Fund. Fangyu Zheng is the corresponding author (*E-mail: zhengfangyu@ucas.ac.cn*).

References

[AB24] Wei Ao and Vishnu Naresh Boddeti. AutoFHE: Automated adaption of CNNs for efficient evaluation over FHE. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2173–2190, Philadelphia, PA, August 2024. USENIX Association.

[BAB+22] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022.

[BBC+25] Loris Bergerat, Charlotte Bonte, Benjamin R Curtis, Jean-Baptiste Orfila, Pascal Paillier, and Samuel Tap. Sharing the mask: Tfhe bootstrapping on packed messages. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):925–971, 2025.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, pages 309–325, 2012.

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

[Cho23] Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 43(3):9–17, 2023.

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

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640. Springer, 2015.

[DMKMS24] Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized fhew bootstrapping using ring automorphisms. In *IACR International Conference on Public-Key Cryptography*, pages 322–353. Springer, 2024.

[FGT21] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. Redsec: Running encrypted discretized neural networks in seconds. *Cryptology ePrint Archive*, 2021.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 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 *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–934. IEEE, 2023.

[FZZ+25] Guang Fan, Mingzhe Zhang, Fangyu Zheng, Shengyu Fan, Tian Zhou, Xianglong Deng, Wenxu Tang, Liang Kong, Yixuan Song, and Shoumeng Yan. Warpdrive: Gpu-based fully homomorphic encryption acceleration leveraging tensor and cuda cores. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1187–1200. IEEE, 2025.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 169–178, 2009.

[GPVL23] Antonio Guimarães, Hilder VL Pereira, and Barry Van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–35. Springer, 2023.

[GS66] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

[JDW+25] Dian Jiao, Xianglong Deng, Zhiwei Wang, Shengyu Fan, Yi Chen, Dan Meng, Rui Hou, and Mingzhe Zhang. Neo: Towards efficient fully homomorphic encryption acceleration using tensor core. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 107–121, 2025.

[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 Transactions on Cryptographic Hardware and Embedded Systems*, pages 114–148, 2021.

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):563–591, 2024.

[MDCL+18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[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 *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 17–28, 2021.

[MS18] Daniele Micciancio and Jessica Sorrell. Ring packing and amortized fhew bootstrapping. *Cryptology ePrint Archive*, 2018.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[SYL+25] Shiyu Shen, Hao Yang, Zhe Liu, Ying Liu, Xianhui Lu, Wangchen Dai, Lu Zhou, Yunlei Zhao, and Ray CC Cheung. Velofhe: Gpu acceleration for fhew and tfhe bootstrapping. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):81–114, 2025.

[WLX+25] Han Wang, Ming Luo, Han Xia, Mingsheng Wang, and Hanxu Hou. Accelerating fhew-like bootstrapping via new configurations of the underlying cryptosystems. *Cryptology ePrint Archive*, 2025.

[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.

[Zam22] Zama. Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists, 2022. https://github.com/zama-ai/concrete-ml.

[Zam24] Zama. TFHE-rs: A pure Rust implementation of the TFHE scheme for boolean and integer arithmetics. https://github.com/zama-ai/tfhe-rs, 2024. Accessed: 2024-01-08.

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):25–63, 2024.

[ZZX+25] Tian Zhou, Fangyu Zheng, Zhuoyu Xie, Wenxu Tang, Guang Fan, Yijing Ning, Yi Bian, Jingqiang Lin, and Jiwu Jing. Ml-cube: Accelerating module-lattice-based cryptography using machine learning accelerators with a memory-less design. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1829–1843, 2025.

A Noise and Failure Modeling

Following [WLX+25] (Theorem 4.2), we treat the *final* LWE decryption error as subgaussian and upper bound the decryption failure probability via a tail bound. Let the decision margin be $t = \frac{q}{8}$. Let σ_{tot} denote the aggregate subgaussian parameter of the final LWE error. We use

$$\sigma_{\text{tot}}^2 = \frac{q^2}{Q_{\text{KS}}^2}\left(\frac{Q_{\text{KS}}^2}{Q^2}\sigma_1^2 + \sigma_2^2 + \sigma_3^2\right) + \sigma_4^2, \tag{14}$$

where

$$\sigma_1^2 = \frac{\pi}{72\Delta^2}\left(3n + 2knN + 12n\Delta^2 + 8kNn\Delta^2\right) + \frac{n(k+r)N\sigma^2}{\Delta^2}, \tag{15}$$

$$\sigma_2^2 = \frac{2\pi kN + 3\pi}{18}, \qquad \sigma_3^2 = kN\,\ell_{\mathrm{KS}}\,\sigma_{\mathrm{KS}}^2, \qquad \sigma_4^2 = \frac{2\pi n + 3\pi}{18}. \tag{16}$$

To refresh two independently bootstrapped ciphertexts c_1, c_2, requiring $|\mathrm{Err}(c_1)+\mathrm{Err}(c_2)| < t$ with $t = q/8$. Since $\mathrm{Err}(c_1) + \mathrm{Err}(c_2)$ is subgaussian with parameter $\sqrt{2}\sigma_{\mathrm{tot}}$, we use

$$p_{\mathrm{fail}} \;\le\; 2\exp\left(-\pi\frac{t^2}{2\sigma_{\mathrm{tot}}^2}\right), \qquad t = \frac{q}{8}. \tag{17}$$

$\sigma = \sigma_{\mathrm{KS}} = \sqrt{2\pi}\cdot 3.19$, and $\Delta = \left\lfloor \frac{T}{Q^2} \right\rfloor$ (so $\Delta \cdot Q^2 \le T$). Under CM packing, r enters only via the $(k+r)$ term in (15).

B Building Blocks for CM-Bootstrapping

This appendix details the procedure of BlindRotate, SampleExtract and KeySwitch for CM packing.

Algorithm 2 CM-Blind Rotate

Require: CM-LWE $\mathbf{c}_{in} = (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^r$ with $\mathbf{b} = (b_1, \ldots, b_r)$; LUT polynomials $\mathbf{v} = (v_1, \ldots, v_r) \in \mathcal{R}_Q^r$; CM-BSK $\{\mathbf{C}_{0i}, \mathbf{C}_{1i}\}_{i=1}^n$
Ensure: CM-GLWE accumulator $\mathsf{ACC} \in (\mathcal{R}_Q^k \times \mathcal{R}_Q^r)$

```
ACC_0 ← (0 ∈ R_Q^k, (X^{-b_1} v_1, ..., X^{-b_r} v_r))
for i = 1 to n do                                   ▷ ⊡ as External Product
    ACC_i ← (X^{a_i} − 1)(ACC_{i−1} ⊡ C_{0i}) + (X^{−a_i} − 1)(ACC_{i−1} ⊡ C_{1i}) + ACC_{i−1}
end for
return ACC_n
```

Algorithm 3 CM-Sample Extract

Require: CM-GLWE $\mathbf{c} = \big((\mathbf{a}_1, \ldots, \mathbf{a}_k), (b_1, \ldots, b_r)\big) \in (\mathcal{R}_Q^k \times \mathcal{R}_Q^r)$; index $\tau = 0$
Ensure: CM-LWE $\mathbf{c}' = (\mathbf{a}', \mathbf{b}') \in \mathbb{Z}_Q^{kN} \times \mathbb{Z}_Q^r$

```
a' ← φ(a_1) ‖ ··· ‖ φ(a_k)                          ▷ φ(·) as in Section 2
b' ← (b_{1,τ}, ..., b_{r,τ})
return (a', b')
```

Algorithm 4 CM-Key Switch

Require: CM-LWE ciphertext $\mathbf{c} = (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}_{q_{\mathrm{KS}}}^{kN} \times \mathbb{Z}_{q_{\mathrm{KS}}}^r$; CM-KSK $\{\mathbf{K}_{i,j,v}\}$; base B_{KS}; length d_{KS}
Ensure: CM-LWE ciphertext $\mathbf{c}_{out} \in \mathbb{Z}_{q_{\mathrm{KS}}}^n \times \mathbb{Z}_{q_{\mathrm{KS}}}^r$ under the target key

```
c_out ← (0 ∈ Z_{q_KS}^n, b)
for i = 0 to kN − 1 do
    for j = 0 to d_KS − 1 do                        ▷ Digit decomposition of a_i in base B_KS
        a_{i,j} ← ⌊a_i / B_KS^j⌋ mod B_KS           ▷ a_i = Σ_{j=0}^{d_KS−1} a_{i,j} B_KS^j
        c_out ← c_out − K_{i,j,a_{i,j}}
    end for
end for
return c_out
```