BEYOND MATRIX ARITHMETIC: RETHINKING LINEAR PROJECTIONS FOR HOMOMORPHIC TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Fully homomorphic encryption (FHE) enables Transformer inference on encrypted data, but its high execution cost limits practical deployment. Existing systems optimize packed computation, GPU kernels, and encoded-weight storage, yet weight preparation remains costly when projection layouts require many encoded operands and redundant expansion. Limited GPU memory further leads to repeated encoding when prepared weights cannot be retained across calls. Motivated by profiling that reveals substantial weight-preparation costs, we rethink linear projection execution in GPU CKKS by jointly organizing layouts, preparation, and retention. Phase-specific plans avoid full-slot weight expansion during prefill and reduce encoded operand counts through rectangular vector–matrix computation during decode, while preserving compatibility with adjacent operators. We separate cross-call retention from current-batch execution: selected complete projection weight sets remain cached, while other weights are prepared in temporary batches without evicting retained sets. Cached and newly prepared weights share a batched accumulation kernel that directly consumes compact encoded operands. On Llama-3-8B with an A100 GPU, 128-token first-layer prefill latency decreases from 1099.3 to 303.7 s, a 72.4% reduction. Across the first three layers at 129 keys, average decode latency is 16.0 s per layer, compared with 65.8 s for a same-backend Cachemir-style integration, including required interface adaptations. Together with component ablations, these results show that reducing weight preparation and preserving encoded-weight reuse are effective complements to optimizing encrypted arithmetic.

1 INTRODUCTION

Cloud-based Transformer inference raises privacy concerns when processing sensitive user data. Fully homomorphic encryption (FHE) enables inference on encrypted data without revealing it to the server. CKKS supports packed approximate arithmetic (Cheon et al., 2017; 2018b), making it suitable for homomorphic Transformer inference. However, ciphertext multiplication, rotation, and bootstrapping impose substantial computational overhead. Linear projections also require weight preparation: plaintext weight matrices must be organized and encoded into operands compatible with the ciphertext layout. This preparation adds to the latency of encrypted computation.

Recent systems accelerate homomorphic Transformer inference through packed matrix algorithms, operator layouts, and GPU kernel optimization. THOR, MOAI, and Cachemir optimize packed computation and data layouts (Moon et al., 2025; Zhang et al., 2026; Yu et al., 2026), while Cheddar provides fused accumulation and Cerium combines compact representations with memory scheduling (Choi et al., 2025; Jayashankar et al., 2025). These techniques improve computation and storage efficiency, but their benefits also depend on how encoded weights are supplied to the computation. Accelerating a matrix kernel does not by itself reduce the cost of generating its operands, and compressing encoded weights does not ensure that they remain available for subsequent calls. Weight preparation and reuse therefore need to be considered together with projection computation.

To quantify these costs, we profile Llama-3-8B (AI@Meta, 2024) on an A100 GPU with a Phantom-based backend. Averaged over the measured layers, linear projections account for 96.0% of 128-token prefill time and 99.1% of decode time at 129 keys. Within these projections, weight expansion, diagonal preparation, and encoding API calls account for 96.3% and 95.7% of wall time, respec-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

tively. These host-side intervals include data movement and synchronization, rather than encoding computation alone. Under the measured blocked representation, compressed encoded weights occupy 62.6 GiB per decode layer, limiting retention alongside cryptographic keys, KV state, and workspace. Although model weights remain unchanged across decode steps, operands that are not retained require repeated preparation. These findings motivate reducing both the cost of preparing weights and the frequency of preparing them again.

Addressing these costs presents **two design challenges**. First, reducing preparation costs requires accounting for their effects on computation and subsequent operators. A projection layout determines the number of encoded operands, the repeated values expanded for each operand, and the conversions required by the next operator. A layout that reduces preparation may therefore introduce additional computation or conversion. Second, retaining encoded weights for reuse reduces the memory available for current execution. The system must decide which weights to retain and how to supply the remaining operands without disrupting reuse or exceeding available GPU memory.

We address the first challenge with phase-specific projection plans that organize weight preparation together with computation and input/output transformations. Prefill prepares each periodic weight diagonal from a single period, avoiding explicit expansion across all slots. Decode uses a rectangular vector–matrix layout to reduce the number of encoded operands relative to the blocked decomposition. Both plans preserve the layouts required by surrounding operators.

To address the second challenge, we separate weight retention across calls from the operands required by the current batch. Selected complete projection weight sets remain cached, while other weights are prepared and consumed in temporary batches without evicting retained sets. Both sources supply the same batched accumulation kernel, which reads compact encoded weights directly. The design reduces preparation through the projection representation and avoids repeated encoding through selective retention, without requiring all weights to remain resident.

Our contributions are summarized as follows:

- We characterize weight preparation and repeated encoding as major costs in the evaluated GPU CKKS Transformer implementation, linking them to projection decomposition, redundant weight expansion, and limited encoded-weight retention.
- We develop a coordinated design for projection preparation and execution. Phase-specific layouts reduce operand generation and expansion, while projection-level retention and temporary batching preserve reuse under GPU memory constraints.
- We evaluate the design on Llama-3-8B with an A100 GPU through layer-level measurements and component ablations. For 128-token prefill, first-layer latency decreases from 1099.3 to 303.7 s, a 72.4% reduction. For decoding at 129 keys across the first three layers, average per-layer latency is 16.0 s, compared with 65.8 s for a same-backend Cachemir-style integration, including required interface adaptations [1].

2 BACKGROUND

2.1 RNS-CKKS SCHEME

CKKS is an FHE scheme for approximate arithmetic over encrypted real or complex vectors (Cheon et al., 2017). For ring degree N, each ciphertext packs up to $N/2$ SIMD slots. Under an active modulus $Q = \prod_{i=0}^{L-1} q_i$, RNS represents each polynomial as L limbs, while NTT converts each limb into a form supporting pointwise modular multiplication (Cheon et al., 2018b). Slots define the logical vector layout, whereas RNS limbs and NTT coefficients form the underlying polynomial representation; rotations permute slots via automorphisms and key switching. Ciphertext multiplication increases scale and component count, followed by relinearization and rescaling to restore a two-component ciphertext and reduce modulus/scale. Bootstrapping refreshes the ciphertext to recover computational depth (Cheon et al., 2018a).

[1] Our implementation is available: `https://anonymous.4open.science/r/GATOR-C46E`

2.2 FHE Transformer Inference

We consider Transformer inference with plaintext model weights and encrypted activations (Moon et al., 2025; Yu et al., 2026). Linear layers, including Q/K/V/O and Llama FFN projections, therefore use ciphertext–plaintext matrix multiplication, whereas $QK^\top$, AV, and FFN gate–up products involve two encrypted operands. Softmax, normalization, and nonlinear activations are implemented with homomorphic approximation circuits (Moon et al., 2025). Autoregressive inference consists of prefill and decode: prefill processes multiple prompt tokens and constructs KV state, while decode processes one new token using previously cached keys and values. Thus, linear projections operate on multiple token rows during prefill but a single feature vector during decode. Model weights are reused across decode steps, whereas activations and KV state evolve; accordingly, the KV cache is distinct from any cache of encoded model weights.

2.3 Homomorphic Matrix Multiplication Optimizations

Diagonal methods implement plaintext-weight linear projections as weighted sums of ciphertext rotations (Halevi & Shoup, 2018):

$$Y_o = \sum_{j \in \mathcal{J}_o} C_{o,j} \odot W^{\mathrm{pt}}_{o,j}, \tag{1}$$

where $C_{o,j}$ is a rotated or aligned input and $W^{\mathrm{pt}}_{o,j}$ is an encoded weight diagonal. BSGS reduces the required rotations, while hoisting amortizes input-dependent key-switching preparation across rotations of the same ciphertext (Halevi & Shoup, 2018; Bossuat et al., 2021). GPU backends further reduce intermediate traffic through kernel fusion and fused multiply–accumulate (MAC) (Yang et al., 2024; Choi et al., 2025). Orthogonally, periodic diagonals can exploit repeated polynomial/NTT structure for compressed storage and indexed consumption (Jayashankar et al., 2025), while caching reusable encoded weights avoids repeated preparation when their representation and encoding state remain compatible.

3 Motivation

3.1 Linear Projection Cost Breakdown

Figure 1(a) profiles Llama-3-8B inference on an A100-SXM4-80GB using a Phantom-based backend. Averaged over the measured layers, linear projections account for 96.01% of 128-token prefill time and 99.11% of decoding time at token 129. Panel (b) breaks down projection wall-clock time into weight expansion, diagonal preparation, encoding API calls, and remaining execution. The measurements reveal substantial weight preparation costs and a large memory footprint that limits encoded-weight reuse.

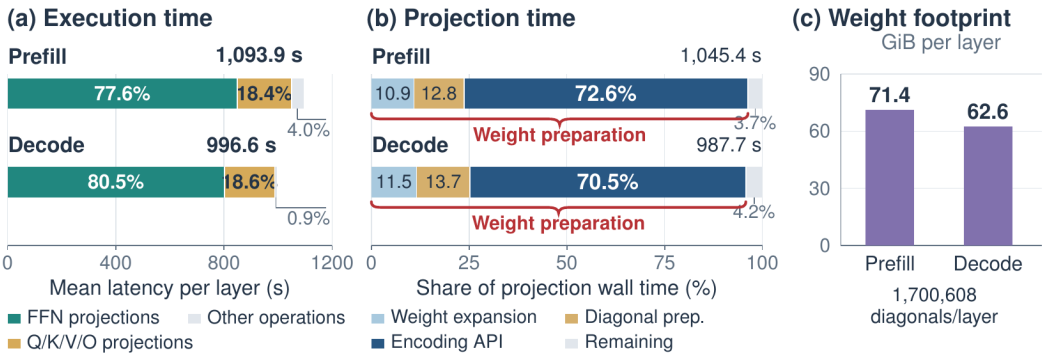

Figure 1: Projection execution and weight-storage requirements. (a) Overall execution profile. (b) Host-side projection-time accounting. Encoding API intervals include synchronization waits. (c) Calculated compressed-weight footprints and recorded reuse under the blocked execution scheme.

Weight preparation dominates projection execution time. Weight preparation comprises weight expansion, per-diagonal preparation, and encoding API calls, together accounting for 96.3% of prefill projection time and 95.7% of decode projection time shown in Figure 1(b). Encoding calls alone contribute 72.6% and 70.5%, respectively; these host-side intervals include data movement and synchronization. The remaining execution, including matrix kernels, accounts for only 3.7% and 4.3%. The measured 256-wide decomposition produces 1,700,608 weight diagonals per Transformer layer. Although each diagonal has a 256-slot period, it is expanded to 32,768 FP32 values before encoding, generating 207.594 GiB of cumulative expansion data per layer. Single-token decoding requires the same number of diagonals. Thus, preparation cost reflects both the large operand count and the explicit replication within each operand.

Limited memory restricts encoded-weight reuse. Model weights remain unchanged across decoding steps, allowing their encoded representations to be reused. However, under the measured blocked representation, compressed encoded weights occupy 71.360 GiB per prefill layer and 62.627 GiB per decode layer. A single layer therefore consumes much of the GPU memory needed for keys, KV state, activations, and temporary workspace, making retention across all layers impractical. Weights that cannot be retained must be encoded again when reused. Reducing preparation cost alone does not eliminate this recurring work, motivating a retention strategy that preserves selected encoded weights while leaving sufficient memory for execution.

3.2 Weight Preparation beyond Operator Optimizations

Existing approaches improve homomorphic matrix computation by reducing the number of homomorphic operations, optimizing ciphertext layouts, and accelerating or fusing operator kernels. THOR and MOAI optimize packed matrix operations, Cachemir specializes vector–matrix multiplication for autoregressive decoding, and Cheddar supports fused accumulation (Moon et al., 2025; Zhang et al., 2026; Yu et al., 2026; Choi et al., 2025). These optimizations primarily reduce the cost of operating on encoded weights. They do not, by themselves, remove the work required to construct and encode those weights. Our measurements identify weight preparation as a major cost in linear projections, especially on a large model. Accelerating the matrix kernel leaves these costs unchanged if the operand generation and encoding path are preserved. Moreover, when encoded weights cannot be fully retained across decoding steps, the preparation work recurs even though the model weights remain unchanged. These opportunities complement reductions in homomorphic operation counts and improvements to operator kernels.

3.3 Opportunity and Challenges

Reduce operand count and redundant expansion. Weight-preparation cost depends on the number of operands to encode and the amount of data expanded for each operand. Periodic diagonals can be encoded from a single period, while single-token decode need not use the same decomposition as prefill. This motivates period-sized preparation and phase-specific projection plans. The challenge is to support matrix accumulation without costly expansion or conversion, while preserving weight indexing, valid-slot masks, output positions, and required ciphertext levels and scales.

Separate persistent retention from temporary execution. Selected projections can retain their complete encoded weight sets, while others encode and consume temporary batches through the same multiply–accumulate implementation. The challenge is to balance memory footprints against the preparation work saved by reuse, reserve space for temporary execution, and prevent temporary batches from evicting retained weights. Both retained and newly encoded operands must meet the required precision, scale, and level constraints.

4 System Design

Figure 2 illustrates the execution of linear projections in FFN gate/up/down and attention Q/K/V/O. Each projection accepts encrypted activations, a plaintext weight matrix, an optional bias, and the required input and output layouts, levels, and scales. It returns the encrypted projection result. Normalization, attention computation, and nonlinear operators remain outside this module.

The system connects two components. **Phase-specific projection layouts** determine how activations occupy ciphertext slots and how the weight matrix is decomposed into diagonal operands. Prefill uses packed token rows with periodic weights, while decoding uses contiguous feature positions and a rectangular matrix decomposition. **Weight preparation and retention** converts these diagonals into compact encoded operands. Selected projections retain their encoded weights across calls; the remaining projections prepare weights in temporary batches. Both paths use the same batched multiply–accumulate implementation. Input and output conversions connect each projection layout to adjacent operators while preserving feature positions and the required ciphertext states.

4.1 Phase-Specific Projection Layouts

A projection layout specifies the placement of encrypted features and the decomposition of the plaintext weight matrix into diagonal operands. These choices determine the number of operands, their repetition periods, and the rotations required for matrix computation. We use different layouts for multi-token prefill and single-token decoding.

Prefill: preserve token parallelism and prepare periodic weights. Prefill uses a 256-feature token-row layout. Each matrix block is evaluated using signed diagonals whose weight pattern repeats across token rows. BSGS organizes input rotations and weighted accumulation (Halevi & Shoup, 2018), followed by group merging and output alignment. This layout preserves parallel computation across tokens. Its weight diagonals have period $p = 256$, allowing the preparation stage to generate one period rather than explicitly repeat it across all slots.

Decode: replace small blocks with a rectangular decomposition. Decoding processes a single token and does not require the token-row organization used in prefill. A plaintext mask preserves valid input features and zeros unused slots. Rotations and additions then replicate the vector into consecutive slot blocks for rectangular vector–matrix multiplication. The rectangular decomposition defines the diagonal operands and rotations used by BSGS. For the Llama-3-8B projections, the resulting weight periods are $p = 4096$ or $p = 16384$. The preparation stage uses these periods to construct the corresponding encoded operands.

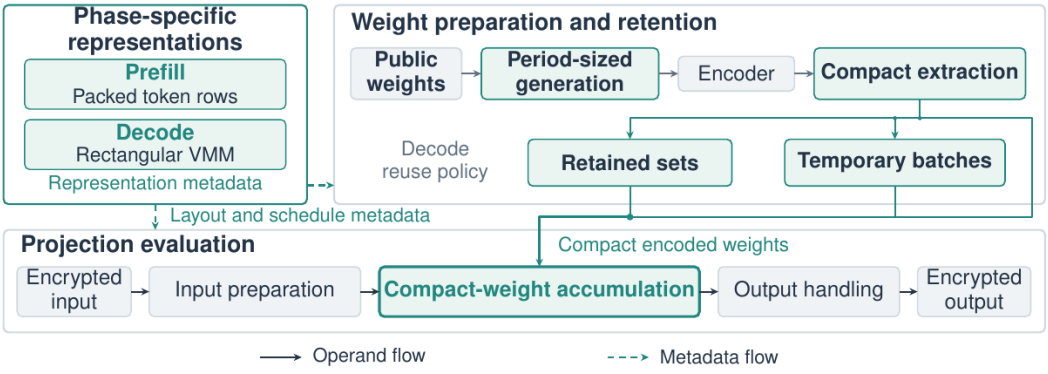

Figure 2: Linear projection architecture.

4.2 Weight Preparation, Retention, and Execution

We separate the decision to retain encoded weights across calls from the batching used to execute the current projection. Retention is managed for complete projection weight sets, while execution requires only the operands of the current batch.

Period-sized weight preparation. The system directly reuses cached operands with matching representations and encoding states. Otherwise, it generates one weight period p, constructs the full RNS/NTT encoding, and extracts $d_p = 2p$ words per active prime under the supported ordering (Jayashankar et al., 2025). New operands are cached only for projections selected for retention; others are consumed in temporary batches.

Projection-level retention. All encoded operands required by a projection under a given layout and encoding configuration form one retention group. A group is selected only if its complete footprint fits within the remaining persistent-cache capacity. The system reserves capacity for the entire group and inserts its operands incrementally as they are generated. Projections that are not selected use temporary batches without evicting retained weights. This prevents temporary preparation from disrupting cross-call reuse.

Shared batched accumulation. Cached and newly prepared operands use the same multiply–accumulate implementation. Within each output block and BSGS group, the system batches at most m compatible ciphertext–weight pairs and applies fused accumulation adapted from Cheddar:

$$A \leftarrow A + \sum_{k \in \mathcal{B}} C_k \odot P_k, \qquad |\mathcal{B}| \le m. \tag{2}$$

Here, C_k is an encrypted activation operand and P_k is its encoded plaintext weight operand. The kernel reads compact weights directly and accumulates products without materializing a separate ciphertext for each product. It follows the required modular arithmetic schedule and writes the accumulator at batch boundaries. Temporary operands are released or their storage is reused after the corresponding GPU computation completes, while retained operands remain available for subsequent calls.

4.3 Cost Model and Execution Constraints

Encoded-weight footprint. Let j identify a projection and its encoding configuration, and let $\mathcal{W}_j(\rho_j)$ denote the operand set defined by representation ρ_j. For operand k, let $p_{j,k}$ denote its period, $d_{j,k} = 2p_{j,k}$ its stored words per active prime, and $L_{j,k}$ its number of active computation primes. With 64-bit words, the encoded-weight payload is

$$B_j(\rho_j) = 8 \sum_{k \in \mathcal{W}_j(\rho_j)} L_{j,k} d_{j,k} \quad \text{bytes.} \tag{3}$$

The representation affects storage through both the number of encoded operands and the storage size of each operand. Retention decisions use the allocated footprint, denoted $\widetilde{B}_j$, which additionally includes metadata, alignment, and other storage overhead.

Repeated preparation cost. For a fixed access sequence, let $g_{j,k}(C)$ be the number of times operand k is generated with persistent-cache capacity C. Let $\hat{t}_{\mathrm{prep}}(j,k)$ be its calibrated preparation time under the required encoding configuration. The estimated preparation cost is

$$\widehat{T}_{\mathrm{prep}}(C) = \sum_{j} \sum_{k \in \mathcal{W}_j(\rho_j)} g_{j,k}(C)\, \hat{t}_{\mathrm{prep}}(j,k). \tag{4}$$

This expression separates three effects: decomposition changes the number of operands, period-sized preparation changes the cost of generating an operand, and retention reduces its repeated generation. Calibration excludes waiting for unrelated earlier GPU work so that synchronization is not mistaken for encoding computation.

Memory and execution constraints. Let $\mathcal{R}$ be the retained projection set. Persistent weights satisfy

$$\sum_{j \in \mathcal{R}} \widetilde{B}_j \le C. \tag{5}$$

During execution, retained weights, temporary batches, and other live data determine peak GPU memory:

$$B_{\mathrm{retained}}(t) + B_{\mathrm{temporary}}(t,m) + B_{\mathrm{other}}(t) \le B_{\mathrm{GPU}}. \tag{6}$$

Here, B_{other} includes keys, activations, KV state, and workspace. Larger batches can reduce kernel launches and accumulator writes but may require more temporary memory. Changing the batch size does not change the operand set. All execution choices must preserve the required output layout, compatible moduli and scales, and numerical accuracy.

5 EVALUATION

5.1 EXPERIMENTAL SETUP

Platform and measurement. Experiments use a Phantom-based CKKS backend on an NVIDIA A100-SXM4-80GB GPU and an Intel Xeon Gold 6338N CPU at 2.20 GHz. We report synchronized wall-clock latency, including online weight preparation and encoding, cache management, encrypted computation, bootstrapping, and required output processing. Model and key initialization, input encryption, warmup, and diagnostic checks are excluded.

Model and workloads. All experiments use Llama-3-8B, with identical weights and inputs across compared methods. We use ring degree $N = 65{,}536$, 32,768 complex SIMD slots, and a 1602-bit ciphertext modulus, targeting 128-bit security following the Homomorphic Encryption Standard (Albrecht et al., 2019). Prefill processes 128 tokens through one Transformer layer. Decoding evaluates the first three layers, with a single query token attending to 129 keys; per-layer averages are reported where indicated. Our method uses the phase-specific weight periods and a batch size of $m = 16$.

Baselines. We implement THOR-style and MOAI-style projection baselines for prefill and a Cachemir-style baseline for decoding on the same Phantom-based backend. All measurements include the interface adaptations required by these implementations. We additionally evaluate ablated variants of our system to isolate the effects of projection layout, weight retention, and batched accumulation. Appendix A.1 details the baseline implementations, cryptographic parameters, and validation coverage.

5.2 MAIN RESULTS

Table 1 reports layer execution times and component breakdowns. Prefill covers the first Transformer layer with 128 tokens; decode starts from a pre-initialized encrypted prefix KV cache and reports per-layer averages at 129 keys. The two workloads are measured separately and do not constitute an end-to-end generation run.

Prefill execution. Our implementation reduces first-layer latency from 1099.3 to 303.7 s, a 72.4% reduction. The seven linear projections decrease from 1056.3 to 260.1 s, while work outside these projections remains approximately 43 s. The layer-level improvement is therefore concentrated in linear projection execution.

Adjusted prefill comparisons. The adjusted first-layer estimates are 937.8 s for THOR-style and 1548.8 s for MOAI-style. Our measured 303.7 s latency is 67.6% and 80.4% lower, respectively. These differences compare measured execution with the stated cost estimates, not new full-system measurements.

Decode execution. At 129 keys, our VMM records 16.0 s per layer, compared with 996.2 s for the historical execution. The matched local CacheMir-style integration takes 65.9 s per layer, giving a $4.12\times$ execution-time ratio. Unlike the adjusted prefill columns, this comparison retains the measured integration boundary, including required layout adapters.

Comparison scope. Prefill configurations and the historical decode record come from separate processes and keys. Separate work-count-based THOR/MOAI projections and assumed parallel-speedup scenarios are reported in Appendix B.3; they are not included as measured full-layer results. The CacheMir-style/VMM comparison shares one context with fixed execution order. These exploratory results characterize the specified execution regions and accounting assumptions, rather than full-model generation accuracy.

5.3 COMPONENT ABLATIONS

Table 2 evaluates projection layout, periodic preparation, weight retention, and batch size. Panel (a) separates changes to the projection representation from changes to operand preparation; panel (b) separates weight retention from batch size. The Q/gate comparisons fix inputs, weights, and key/context within each projection, with caching disabled and $m = 16$.

Projection representation. The first two rows of Table 2(a) compare blocked and rectangular representations with periodic preparation enabled in both. Q projection time decreases from 13.6 to

Table 1: Llama-3-8B execution breakdown (s).

(a) Prefill: one layer, 128 tokens

Region	Baseline	Ours	THOR-style (adjusted)	MOAI-style (adjusted)
Q projection	85.5	19.9	70.5	115.5
K projection	23.7	5.0	23.5	28.9
V projection	20.2	5.0	23.5	28.9
O projection	87.0	20.1	70.5	115.5
FFN gate	295.4	70.1	234.9	404.2
FFN up	278.4	70.3	234.9	404.2
FFN down	266.1	69.7	232.2	404.2
Other time	43.0	43.7	47.7	47.4
Total	**1093.9**	**303.7**	**937.8**	**1548.8**

(b) Decode: layer mean, 129 keys

Region	Baseline	Ours	CacheMir-style
Q projection	74.3	0.8	4.7
K projection	18.4	0.2	3.1
V projection	18.4	0.2	3.2
O projection	74.4	0.8	3.7
FFN gate	268.2	2.2	10.8
FFN up	266.9	2.2	10.6
FFN down	266.7	2.4	10.6
Other time	8.9	7.2	19.2
Total	**996.2**	**16.0**	**65.9**

Projection times include preparation, encoding, and input/output handling; other time includes all remaining execution time within the measured region, excluding the listed matrix multiplications. THOR/MOAI prefill entries use adjusted accounting.

Table 2: Conditional effects of projection components.

(a) Representation and preparation

Controlled change	Before (s)	After (s)	Red. (%)
Representation: periodic inputs, no cache			
Q: blocked → rectangular	13.6	0.9	93.7
Gate: blocked → rectangular	49.8	2.3	95.4
Preparation: full-slot → periodic			
Q projection, blocked	54.5	13.6	75.1
Gate projection, blocked	216.2	49.8	77.0
Complete-region preparation controls			
Prefill layer, 128 tokens	1088.6	298.6	72.6
Decode layer mean, 130 keys	22.7	16.0	29.7

(b) Retention and batching

Retention: 130 keys, per-layer means Configuration	Hits	Encodings	Time (s)
No cache	0	22,528	16.0
Entrywise LRU	0	22,528	16.3
Projection retention	10,240	12,288	14.2
Batching: 129 keys, per-layer means Batch size	Terms	Launches	Time (s)
$m = 1$	22,528	22,528	16.1
$m = 16$	22,528	1,408	16.2

Q/gate rows measure individual projections; prefill and decode all measure one layer. Reductions are computed before rounding. Different sequence positions and cohorts are not pooled.

Table 3: Encoded-weight storage and active memory (GiB).

	Calculated full-set storage		Measured at 129 keys	
Method	Payload	Allocation	Resident weights	Active-pool peak
Ours	105.4	105.4	22.9	43.1
CacheMir-style	105.4	348.0	24.0	51.4

Calculated columns cover all 21 projection weight sets in the three-layer workload, assuming full retention. Measured columns describe the first decode call; full-set allocation is not actual resident memory.

0.9 s (93.7%), and gate projection time from 49.8 to 2.3 s (95.4%). Encoding and product counts fall from 130,816 to 4,096 for Q and from 457,856 to 4,096 for gate. These reductions concern the complete representation-specific projection, including its rotations and input/output processing, rather than encoding alone.

Periodic weight preparation. Under the fixed blocked layout, preparing one period instead of a full-slot vector reduces Q and gate latency by 75.1% and 77.0%, respectively. Generated values decrease by $128\times$, while encoding counts, product terms, and MAC launches remain unchanged. The savings therefore arise from the preparation path, independently of layout changes, caching, or batch size. Separate layer-level measurements show corresponding latency reductions of 72.6% for first-layer prefill and 29.7% for uncached decoding.

Projection-level weight retention. We compare retention policies at the same 130-key call, following each configuration's preceding 129-key call. Under the same cache limit, projection-level retention records an average of 10,240 hits per layer, whereas entrywise LRU records none. Encoding events decrease from 22,528 to 12,288 per layer, and mean latency falls from 16.3 to 14.2 s (13.2%). The no-cache control takes 16.0 s per layer, giving an 11.1% reduction with projection-level retention. These comparisons measure subsequent-call reuse, complementing the first-call results in Section 5.2.

Common consumption path. Cached and freshly encoded weights use the same compact-weight consumer. Increasing m from 1 to 16 reduces launches from 22,528 to 1,408 per layer, while retain-

ing 22,528 product terms. Mean per-layer latency changes from 16.1 to 16.2 s, without an observed improvement in this single comparison.

5.4 Resource Costs

Encoded-weight storage. Table 3 distinguishes calculated full-set storage from measured GPU memory use. Under the evaluated encoding configurations, both implementations require the same active weight payload. The local CacheMir-style port has a larger full-set allocation because each plaintext reserves all 32 limbs, including inactive ones. This difference reflects the port's allocation format, not an intrinsic storage requirement of its matrix algorithm.

Runtime memory. Resident weights occupy only part of the active-pool peak, which also includes context/key allocations, KV state, activations, and workspace. These peaks describe the measured execution footprint, not incremental scratch or per-layer memory. Cache-policy effects on encoding counts and latency are reported separately in Section 5.3.

5.5 Execution Fidelity

In same-key tests of Llama-3-8B decode projections, cache reuse, bypass execution, and batching preserve ciphertext payloads and metadata under identical inputs, VMM representation, and encoding state. All tested linear projections outputs satisfy their registered error requirements (Appendix B.2). These checks validate the tested retention and consumption paths, not the blocked-to-rectangular performance controls in Section 5.3, which do not include numerical validation.

6 Related Work

GPU-based encrypted Transformer inference. NEXUS compresses ciphertexts and folds slots (Zhang et al., 2025); THOR uses diagonal-major evaluation (Moon et al., 2025); MOAI coordinates layouts and batching (Zhang et al., 2026); EncryptedLLM studies GPU FHE and activation approximations (De Castro et al., 2025). We emphasize phase-specific linear projections, including preparation and compatibility with subsequent operators.

Phantom optimizes GPU arithmetic (Yang et al., 2024), Cheddar fuses accumulation (Choi et al., 2025), Cerium combines periodic compression and memory scheduling (Jayashankar et al., 2025), and AEGIS distributes Transformer/RNS execution (Gong et al., 2026). Our design selectively retains complete projection weight sets while preparing nonretained weights in temporary batches for the same consumer. Cachemir co-designs interleaved VMM, extraction fusion, KV layouts, and bootstrap placement (Yu et al., 2026). Our decode instead uses continuous-feature VMM compatible with shared backend interfaces, coordinated with selective encoded-weight caching.

Other execution frameworks and privacy models. Fhelipe automates FHE packing (Krastev et al., 2024); ARION fuses attention in Lattigo (Yang et al., 2025). For unencrypted inference, Corsair-based fusion overlaps normalization and matrix work (Salmani & Soloveychik, 2025), while FlexGen schedules GPU–CPU–disk offloading (Sheng et al., 2023). Puma and BumbleBee provide three-party and two-party Transformer inference, respectively (Dong et al., 2023; Lu et al., 2025). Our focus is projection preparation and reuse under operator-interface and memory constraints.

7 Conclusion

Encoded-weight preparation and reuse remain important costs in GPU CKKS Transformer execution. Projection representations determine preparation work, while current-batch execution does not require persistent retention of an entire projection's encoded weights. Our design combines phase-specialized representations and periodic preparation with selective retention in decode, supplying nonretained weights through temporary batches. Exploratory A100 measurements of Llama-3-8B show 72.4% lower first-layer prefill latency at 128 tokens than the bootstrap-accelerated baseline and a $4.12\times$ ratio between the local CacheMir-style and our mean per-layer decode times at 129 keys; these regional execution-cost comparisons include required adapters and do not establish full-model numerical fidelity.

AI USE STATEMENT

AI assistance was used to draft and revise manuscript text and improve its organization. Reported measurements were obtained from the authors' implementations and experiments, not generated by a language model. The authors are responsible for the technical content, experimental evidence, and final manuscript.

REPRODUCIBILITY STATEMENT

Experiments use Llama-3-8B on an NVIDIA A100-SXM4-80GB GPU with a Phantom-based CKKS backend. Section 5.1 describes the platform, workloads, and measurement scope, while Appendix A.1 provides cryptographic parameters, implementation details, and validation coverage. Reported latencies include online weight preparation, encoding, cache operations, encrypted computation, bootstrapping, and required output handling. Model and key initialization, input encryption, warmup, and diagnostics are excluded.

The evaluation covers 128-token first-layer prefill and single-token decoding across the first three layers, with prefix KV states computed on the CPU and encrypted separately. THOR-style, MOAI-style, and Cachemir-style comparisons use implementations within the same backend, including required interface adaptations, rather than measurements of the original systems. Component experiments specify their controlled variables and measurement scope in Section 5.3. Projection-level timing controls use one observation per configuration without numerical validation; full-model accuracy and end-to-end generation are not assessed. The implementation is available in an anonymous repository.[2]

ETHICS STATEMENT

This work studies the execution efficiency of homomorphic Transformer inference with encrypted activations and plaintext model weights. It aims to reduce computational costs while retaining the confidentiality protections of the underlying FHE scheme under its stated security assumptions. It does not provide model-weight confidentiality or additional protection against information revealed by model outputs, access patterns, or implementation side channels. The proposed optimizations do not address harmful model behavior or misuse. Released artifacts should exclude secret keys, credentials, private input data, and identifying infrastructure details.

REFERENCES

AI@Meta. Llama 3 model card, 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption standard. Cryptology ePrint Archive, Paper 2019/939, 2019. URL https://eprint.iacr.org/2019/939.

Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Advances in Cryptology – EUROCRYPT 2021*. Springer, 2021. doi: 10.1007/978-3-030-77870-5_21. URL https://eprint.iacr.org/2020/1203.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, pp. 409–437. Springer, 2017. doi: 10.1007/978-3-319-70694-8_15. URL https://eprint.iacr.org/2016/421.

[2] https://anonymous.4open.science/r/GATOR-C46E

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2018*, pp. 360–384. Springer, 2018a. doi: 10.1007/978-3-319-78381-9_14. URL https://eprint.iacr.org/2018/153.

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2018/931, 2018b. URL https://eprint.iacr.org/2018/931.

Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library designed for GPU architectures. arXiv:2407.13055v2, 2025. URL https://arxiv.org/abs/2407.13055v2. Accepted at ASPLOS 2026.

Leo De Castro, Daniel Escudero, Adya Agrawal, Antigoni Polychroniadou, and Manuela Veloso. EncryptedLLM: Privacy-preserving large language model inference via GPU-accelerated fully homomorphic encryption. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 12677–12688. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/de-castro25a.html.

Ye Dong, Wen jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, and Wenguang Chen. PUMA: Secure inference of LLaMA-7B in five minutes. arXiv preprint arXiv:2307.12533, 2023. URL https://arxiv.org/abs/2307.12533.

Zhaoting Gong, Ran Ran, Fan Yao, and Wujie Wen. Aegis: Scaling long-sequence homomorphic encrypted transformer inference via hybrid parallelism on multi-gpu systems, 2026. URL https://arxiv.org/abs/2604.03425.

Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. Cryptology ePrint Archive, Paper 2018/244, 2018. URL https://eprint.iacr.org/2018/244.

Siddharth Jayashankar, Joshua Kim, Michael B. Sullivan, Wenting Zheng, and Dimitrios Skarlatos. A scalable Multi-GPU framework for encrypted large-model inference. arXiv preprint arXiv:2512.11269, 2025. URL https://arxiv.org/abs/2512.11269.

Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656382. URL https://doi.org/10.1145/3656382.

Lattice Estimator contributors. Lattice Estimator: Security estimates for lattice problems. Software repository, 2026. URL https://github.com/malb/lattice-estimator/tree/53da5982597709ba0fdf94ea37a84d822310fd84. Commit 53da5982597709ba0; accessed September 19, 2026.

Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and Wenguang Chen. BumbleBee: Secure two-party inference framework for large transformers. In *32nd Annual Network and Distributed System Security Symposium (NDSS 2025)*. The Internet Society, 2025. doi: 10.14722/ndss.2025.230057. URL https://www.ndss-symposium.org/ndss-paper/bumblebee-secure-two-party-inference-framework-for-large-transformers/.

Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. THOR: Secure transformer inference with homomorphic encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3765–3779. Association for Computing Machinery, 2025. doi: 10.1145/3719027.3765150. URL https://doi.org/10.1145/3719027.3765150.

Mahsa Salmani and Ilya Soloveychik. LLM inference acceleration via efficient operation fusion. arXiv preprint arXiv:2502.17728, 2025. URL https://arxiv.org/abs/2502.17728.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/sheng23a.html.

Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. Phantom: A CUDA-accelerated word-wise homomorphic encryption library. *IEEE Transactions on Dependable and Secure Computing*, 21(5):4895–4906, 2024. doi: 10.1109/TDSC.2024.3363900. URL https://eprint.iacr.org/2023/049.

Linhan Yang, Jingwei Chen, Wangchen Dai, Shuai Wang, Wenyuan Wu, and Yong Feng. ARION: Attention-optimized transformer inference on encrypted data. Cryptology ePrint Archive, Paper 2025/2271, 2025. URL https://eprint.iacr.org/2025/2271.

Ye Yu, Yifan Zhou, Yi Chen, Pedro Soto, Wenjie Xiong, and Meng Li. Cachemir: Fully homomorphic encrypted inference of generative large language model with KV cache. arXiv preprint arXiv:2602.11470, 2026. URL https://arxiv.org/abs/2602.11470.

Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. In *Network and Distributed System Security Symposium (NDSS)*, 2025. URL https://www.ndss-symposium.org/ndss-paper/secure-transformer-inference-made-non-interactive/.

Linru Zhang, Xiangning Wang, Jun Jie Sim, Zhicong Huang, Jiahao Zhong, Huaxiong Wang, Pu Duan, and Kwok Yan Lam. MOAI: Module-optimizing architecture for non-interactive secure transformer inference. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=qJn4HtTzhH.

A LLAMA-3-8B CONFIGURATION AND VALIDATION

A.1 CRYPTOGRAPHIC CONFIGURATION

The Llama-3-8B comparisons use $N = 65{,}536$, 32,768 slots, secret weight 192, CBD(21) errors, and application scale approximately 2^{40}. The ordered Q25/P7 bases in Table 4 follow Cheddar's reference configuration (Choi et al., 2025); execution retains the common Phantom-based bootstrap backend. The full modulus has $\log_2(QP) = 1602.0003$. Fresh ciphertexts start at $L = 25$; bootstrap enters at $L = 1$, raises to 25, and completes CtS/EvalMod/StC at $L = 22/14/11$. The seven special primes do not add computation depth.

Table 4: Exact ordered Q25/P7 primes used by the Llama-3-8B runs (zero-based indices). Read each basis by index; the two columns only shorten the listing.

Index	Prime	Index	Prime
q_0	1125899908022273	q_{16}	36028797033644033
q_1	1099515691009	q_{17}	36028797037576193
q_2	1099523555329	q_{18}	36028797048324097
q_3	1099525128193	q_{19}	36028797048586241
q_4	1099526176769	q_{20}	36028797049896961
q_5	1099529060353	q_{21}	36028797051863041
q_6	1099535220737	q_{22}	36028797053698049
q_7	1099536138241	q_{23}	36028797054222337
q_8	1099537580033	q_{24}	35184372744193
q_9	1099538104321	p_0	72057594038321153
q_{10}	1099540725761	p_1	72057594040680449
q_{11}	36028797019488257	p_2	72057594042646529
q_{12}	36028797023420417	p_3	72057594047889409
q_{13}	36028797024206849	p_4	72057594057195521
q_{14}	36028797025124353	p_5	72057594058375169
q_{15}	36028797032202241	p_6	72057594058899457

The bootstrap uses cosine degree 63, $K = 25$, loge 10, two double-angle steps, and inverse degree one. Integer amplification by four and inverse compensation in the final StC plaintext preserve the registered rescale schedule. Source revisions, coefficients, and build commands are in the accompanying artifact index.

Security scope. The registered lattice-estimator screen (Lattice Estimator contributors, 2026) uses GSA and MATZOV classical full-ROP costs for uSVP, BDD, hybrid, and hybrid-MITM attacks. Its minimum 130.4345 bits minus the 1.6152-bit sampler allowance gives 128.8194 bits. ADPS16 core-SVP gives 106.288 bits under a different cost model. The implemented sign probabilities are 86/171 and 85/171, and support selection differs from the ideal sparse distribution; actual sparse-secret dual attacks are unsupported by the estimator. The primal screen therefore does not establish unconditional 128-bit security.

B PROJECTION EXECUTION INVARIANTS AND ACCOUNTING

B.1 COMPACT REPRESENTATION AND MAC CORRECTNESS

Compact representation. For a supported weight period p, the encoder first constructs the full RNS/NTT representation of the plaintext weight under the required modulus state. Compact extraction retains $d_p = 2p$ coefficients per active prime. For coefficient j, the extracted representation stores the value corresponding to position jN/d_p in the original ordering, and the consumer maps coefficient r to compact index

$$\lfloor r/(N/d_p) \rfloor .$$

The prefill representation uses $p = 256$, whereas the evaluated decode rectangular VMM uses $p = 4096$ or $p = 16384$. The compact representation reduces retained storage and changes operand access, while the full RNS/NTT construction remains part of the preparation path.

Table 5: Llama-3-8B projection validation. Maximum absolute errors over evaluated cache, bypass, and batched-consumption paths.

Projection	Max. absolute error	Validation
Q	4.61×10^{-7}	PASS
K	3.32×10^{-6}	PASS
V	4.03×10^{-6}	PASS
O	5.42×10^{-7}	PASS
Gate	5.30×10^{-7}	PASS
Up	6.57×10^{-7}	PASS
Down	4.69×10^{-6}	PASS

Batched MAC correctness. The batched projection consumer applies the same modular accumulation recurrence as the single-term execution. For active prime q_i, ciphertext coefficient c_k, and compact weight coefficient w_k,

$$u_k = \mathrm{canon}_{q_i}(c_k w_k), \qquad a_{k+1} = \mathrm{canon}_{q_i}(a_k + u_k).$$

Each product and addition is reduced immediately. Because $q_i < 2^{63}$, the addition of two canonical residues remains representable in the 64-bit residue domain. The batched kernel preserves the original term order and recurrence while writing the accumulator only at batch boundaries. Therefore, induction over the accumulated terms gives the same modular result as the corresponding single-term execution for identical operands and states.

This argument applies only to fixed operands, term ordering, and state transitions. It does not move CKKS rounding operations, change rescale placement, or prove equivalence between different projection representations.

Implementation validation. Before execution, the wrapper validates the compatibility conditions required for compact consumption, including context and parameter identity, active modulus basis, operand scale, NTT representation, compact capacity, stream compatibility, and output nonaliasing.

Retained and newly generated operands are verified to enter the same projection consumer path. Numerical validation of the resulting Llama-3-8B projection and attention operators is reported separately in Appendix B.2.

B.2 Numerical Validation of Optimized Operators

This appendix reports detailed numerical validation for the optimized operators evaluated in Section 5.5. The goal is to verify that representation changes, weight retention, and execution reorganization preserve the required operator-level numerical contracts. These experiments validate individual encrypted operators and do not establish full-model Llama-3-8B generation accuracy.

Llama-3-8B projection validation. We validate the optimized linear projections at the operator boundary. For each Q/K/V/O and FFN gate/up/down projection, encrypted inputs are generated from the same plaintext reference and evaluated through different execution paths, including newly prepared weights, retained weights, bypass execution, and batched consumption. The reference is the corresponding plaintext projection under the registered model weights.

Table 5 summarizes the maximum observed absolute errors. All evaluated outputs satisfy the registered projection error requirement. For compared cache and batching paths, output payloads and metadata are identical, demonstrating that weight retention and batched consumption do not change the projection result under the same representation and state.

Llama-3-8B attention-consumer validation. The attention validation compares blockwise consumption, aggregate placement, and factorized consumption against the same plaintext attention reference. Unlike projection caching, these transformations reorder encrypted operations and therefore are not expected to be bitwise identical.

Table 6 reports the maximum observed AV and output-projection errors. All evaluated paths satisfy the module-level tolerance of 10^{-3}. The factorized consumer changes the execution order while preserving the required weighted-sum computation, which is validated numerically rather than through ciphertext equality.

Table 6: Llama-3-8B attention-consumer validation. Errors are measured against the standard attention reference.

Consumer	Max. AV error	Max. O error
Blockwise	4.75×10^{-5}	6.96×10^{-6}
Placement	4.53×10^{-5}	6.97×10^{-6}
Factorized	9.55×10^{-6}	2.84×10^{-6}

Table 7: Work-count-based projection estimates. Times are seconds; s is an assumed speedup factor.

Method	Projection	Target count	s	Estimate
THOR-style	Q/O	101,376	1	70.5
THOR-style	Gate/up	337,920	1	234.9
MOAI-style	Q/O	16,777,216	8	115.5
MOAI-style	Gate/up/down	58,720,256	8	404.2

Supplementary FFN validation. Additional FFN validation is performed on GPT-2 Base to separate the effect of batched accumulation from upstream Transformer errors. The experiment evaluates the complete FFN path (up projection, GELU approximation, and down projection) with identical encrypted inputs under different accumulation batch sizes.

Both batch configurations produce identical intermediate payloads and metadata. Across three development inputs, the maximum absolute error against the GELU reference remains below 2×10^{-5}, with relative ℓ_2 errors below 2.5×10^{-7}. These results provide supplementary evidence that batched accumulation preserves the FFN computation, but do not represent Llama-3-8B end-to-end accuracy validation.

B.3 Work-Count-Based Prefill Cost Estimates

These estimates extrapolate historical up-projection timings to Llama-3-8B projection workloads. They are distinct from the directly measured execution results.

Calibration and extrapolation. The THOR-style calibration averages three first-use up-projection measurements: 7.797911125, 7.663884028, and 8.031660847 s. Their mean is 7.831152 s for 11,264 native weight multiplications. The MOAI-style calibration uses one up-projection measurement of 129.9188333 s for 2,359,296 scalar multiplications. The THOR calibration covers the up projection, not the GELU activation.

For each method, target cost is estimated as

$$\widehat{T}_{j,\mathrm{native}}(s) = t_{\mathrm{cal}} \frac{n_j}{n_{\mathrm{cal}}} \frac{1}{s}, \tag{7}$$

where t_{cal} and n_{cal} are the calibration time and work count, and n_j is the registered target count in the same method-specific units. The factor s denotes an assumed effective speedup relative to the unadjusted prediction.

Parallel-speedup assumption. The selected MOAI scenario divides the native-region prediction by $s = 8$. This is an assumed effective speedup, not a measured OpenMP improvement, a thread-count conversion, or batch amortization. No division by 256 is applied.

Scope and limitations. Residual interface cost A_j is unmeasured and excluded from Table 7. A complete projection estimate would require $\widehat{T}_{j,\mathrm{native}} + A_j$. Calibration and target configurations differ in shape, active modulus state, cache conditions, and execution boundaries. The estimates have not been validated against complete target-shape measurements and do not establish full-layer latency or a performance bound.

B.4 Layout, State, and Cache Lifetime

The decode interface accepts one vector in consecutive slots. It masks the valid input, replicates it for rectangular BSGS, evaluates the projection, folds partial outputs when needed, and masks the output. Input dimensions are padded to powers of two; the 14,336-wide FFN uses 16,384 internally. Input

masking, matrix rescale, and output masking each consume one level: Q/K/V follow $L = 8 \to 5$ and the remaining projections follow $L = 11 \to 8$. These transitions use actual arithmetic. The token-row prefill layout has its own adapter and state schedule.

Projection admission reserves the complete encoded-weight footprint on first encounter if it fits the remaining budget. Accepted sets are filled as their weights are generated and retained across decode steps; rejected sets use temporary batches without evicting retained projections. The scope binds context, immutable weight identity, dimensions, level, and exact encoding/input scales. Weight replacement or context teardown requires synchronization and cache clearing before addresses can be reused. Temporary payloads remain alive through submission on the common CUDA stream; the projection synchronizes before returning. This deterministic first-fit policy does not optimize the retained subset.

B.5 Timing and Memory Accounting

Decode timing boundaries. The decode results in Table 1 report complete measured regions rather than isolated matrix kernels. For the matched 129-key comparison, the local CacheMir-style integration spends on average 53.253 s per layer in projection input/output handling and QKV-to-attention conversion, whereas the corresponding measured cost is 0.188 s for our continuous VMM. Their native matrix intervals, including weight preparation, average 5.260 and 8.581 s per layer, respectively. Thus, the complete integration comparison reflects both matrix execution and the interfaces required by each local implementation. These intervals are accounting boundaries, not independently toggled causal effects, and the measured adapter cost of the local port is not an intrinsic requirement of the original CacheMir system.

Encoded-weight storage. Table 3 distinguishes active weight payload, allocated capacity, persistent residency, and the CUDA active-pool high-water mark. Across the evaluated three-layer decode workload, both implementations contain 105.375 GiB of active encoded-weight payload. The local CacheMir-style port allocates 348 GiB for the corresponding full weight set because its plaintext objects reserve inactive limbs; this reflects the evaluated storage format rather than an algorithmic lower bound. Neither full-set value represents actual resident memory.

The measured runs limit persistent encoded-weight residency to 24 GiB as an experimental configuration. This is not a device-capacity limit: active-pool peaks also include context and evaluation keys, encrypted KV state, activations, temporary operands, and workspace. Consequently, calculated full-set capacity, resident encoded weights, and process memory peaks describe different quantities and should not be interchanged.

The retention-policy effects on encoding counts and latency are reported in Section 5.3. No controlled cache-capacity sweep is reported. The decode implementation uses $m = 16$ by default; the separate $m = 1$ versus $m = 16$ control measures batching granularity but does not establish an optimal batch size.

C Execution Invariants and Supporting Controls

C.1 Compressed Weights and MAC Correctness

A missing period-p diagonal is encoded into a full $L \times N$ RNS/NTT plaintext before extracting $d_p = 2p$ coefficients per prime. Extraction reads coefficient jN/d_p; the consumer maps coefficient r to compact index $\lfloor r/(N/d_p) \rfloor$. Cached weights bind shape, period, BSGS group, ordered primes, exact scale, and weight revision. Multi-term MAC requires resident GPU payloads; a miss is handled by the separate preparation path.

Each product is canonical. Since $q_i < 2^{63}$, adding two canonical residues fits in 64 bits; conditional subtraction restores $[0, q_i)$. Induction over terms establishes equality with the single-term accumulator after every update. Writing only at batch boundaries therefore preserves group results. Original group merges, rotations, rescale, and bias remain; no rounding operations are combined.

The wrapper checks context, parameter/scale compatibility, component count, NTT form, period/capacity, stream, and nonaliasing destinations before launch. Rejections preserve inputs and existing accumulators. The current mature-MAC fixture passes 51 checks, including 17-term tails

and near-$q_i - 1$ residues. Logical accumulator savings do not establish measured DRAM traffic or per-method peak-memory reductions.

C.2 Exact-Length Query Reuse

Let q_0 be the masked, rescaled query prefix and s the local-token stride. For each distinct valid length ℓ, initialize $P_\ell = q_0$ and execute $P_\ell \leftarrow P_\ell + \rho_{-ts}(q_0)$ for $t = 1, \ldots, \ell - 1$ in the original order. The rotated operand is always q_0, not the running sum. Equal-length blocks read the stored P_ℓ without mutation; different lengths have separate replicas. The map expires at the end of the QK call.

The wrapper checks query shape, two-component NTT form, minimum level, power-of-two head width, slot capacity, and lengths 1–32; backend arithmetic requires compatible contexts and states. Identical masking, rotation, addition, and state history produce identical per-length operands for the unchanged key products and score masks.

C.3 Historical Controls and Evidence Boundary

Historical Q31/P7 projection confirmations use 24 independent process/key pairs per shape; Q37/P7 FFN confirmation uses six. Their intervals apply Student-t to paired log ratios. Those contexts fall below the registered 128-bit target under the recorded MATZOV estimates; their timings remain capability-adoption evidence, separate from current Q25/P7 results.

Against 256-slot pre-expansion, streaming gives only a 0.53% first-use down-projection reduction; the other first/hit comparisons establish no increment. In three independent Graphs blocks, batching retains a 39.16% replay reduction under the shared capture adaptation. A one-sample batch scan finds $m = 32$ faster than the preregistered $m = 16$, without establishing an optimum. The historical implementation passes 115 boundary assertions across levels, scales, periods, tails, and invalid calls.

Earlier slotwise attention has a 3.001141 fixed-circuit/standard-AV discrepancy; its factor execution also fails the 10^{-3} limit on the model-derived fixture. Historical native execution fails numerical/domain checks. These failures remain archived and do not describe the current row-normalized port. Full sample tables, traces, failed runs, and extended schedules are retained in the accompanying artifact records rather than repeated here.