

Terrazzo: Memory-Aware GPU Framework for Private Transformer Inference

Rostin Shokri and Nektarios Georgios Tsoutsos

University of Delaware
{rostinsh, tsoutsos}@udel.edu

Abstract. Fully homomorphic encryption (FHE) allows a server to run inference directly on encrypted data, making it a promising foundation for private transformer inference. Its dominant scheme, CKKS, has no native matrix multiplication, so the encrypted matrix multiplications at the heart of transformers dominate inference cost. The recent GL scheme supports matrix multiplication natively, but its ciphertexts, plaintexts, and evaluation keys are so large that a direct GPU implementation would require terabytes of VRAM for even small language models. We present Terrazzo, a GPU framework for the GL scheme, co-designed across cryptography, algorithms, and kernels to fit private inference on a commodity GPU. Terrazzo bootstraps into a compact Y-decoded representation, so nonlinear layers never hold full ciphertexts live; tiles all work over GL’s independently schedulable ciphertext slices, sizing each tile per level against an L2 and occupancy model; confines grafting-based modulus management to application levels around a sprout-free bootstrap; and keeps every model weight at its cleartext footprint with an expansion-free plaintext pipeline. On BERT-base at full 256-input occupancy, Terrazzo achieves a 7.04s amortized time per input on a consumer 32 GB RTX 5090. Its A100 time is 29.98s amortized, a 2.20–20.09 $\times$ speedup over prior single-A100 systems. Per-slice bootstrap speedup is 1.12–2.03 $\times$ on the RTX 5090 and 1.08–1.36 $\times$ on the A100.

Keywords: Fully homomorphic encryption, GL scheme, private inference, GPU acceleration, transformers, bootstrapping.

1 Introduction

The widespread deployment of Large Language Models (LLMs) has accelerated the demand for private inference. The models that cloud providers serve are private intellectual property, while the data users send them is often sensitive, including medical records, financial documents, and source code. Private inference (PI) targets exactly this tension: the provider does not reveal its model weights, the user does not reveal their data, and inference happens anyway. Among the proposed solutions, fully homomorphic encryption (FHE) offers the simplest deployment model: the client encrypts its input, a single server evaluates the model directly on the ciphertext, and only the client can decrypt the returned result, with no interaction and no trusted parties.

While interactive privacy solutions such as secure multi-party computation (MPC), or hybrid solutions that use both MPC and FHE, are a popular research direction, their massive client-server communication overhead was traditionally justified by the prohibitive cost of server-side FHE bootstrapping. Recent GPU acceleration has made ciphertext bootstrapping cheaper than sending the ciphertext back to the key holder for decrypt and resend: Cheddar [9] reports a 22.1 ms complex slot CKKS bootstrap on an RTX 5090, compared with more than 40 ms merely to transmit a 5 MB ciphertext over a 1 Gbps network. A follow-up work called Theodosian [10] improved on Cheddar with L2-aware keyswitching and algorithmic optimizations, achieving 12.8 ms on the same hardware. These results demonstrate that FHE bootstrapping has reached a practical stage where non-interactive solutions can become the best option for private inference.

The limitation of existing CKKS-based private inference lies in the scheme itself. CKKS has no native matrix multiplication: every matmul must be assembled from slot rotations and diagonal-packed plaintexts, so the dense ciphertext–ciphertext and ciphertext–plaintext products that form the backbone of transformer

models dominate end-to-end cost. The recently proposed GL scheme [18] removes this mismatch by supporting matrix multiplication as a native homomorphic operation, making it a natural candidate for practical private transformer inference. The obstacle is memory: GL ciphertexts, plaintexts, and evaluation keys are orders of magnitude larger than their CKKS counterparts, and a direct GPU port would require terabytes of VRAM to run even small language models.

In this work we present Terrazzo, a GPU framework for the GL scheme, tailored for efficient, low-memory private transformer inference. Our contributions are as follows:

A novel bootstrap workflow with Y-decoded intermediate representation: Standard GL bootstrapping [11, 47] outputs the ciphertext in the canonical three-axis encoding, where a nonlinear layer’s BSGS polynomial evaluation must hold roughly 30 GB of ciphertexts live at once (Section 3.4). Our observation is that the Y-axis DFT is strictly required only by CCMM, whose encrypted operands cannot absorb a change of basis; PCMM can fold any Y-axis encode or decode into its public weight matrix. Thus, it runs between any pair of representations, including from decoded to decoded, and nonlinear layers do not need the Y-axis DFT at all. Therefore, ahead of nonlinear work, we remove the Y-axis DFT and run the bootstrap itself in the decoded representation. In our BERT schedule, the required Y-axis decodes are fused into adjacent weight or padding plaintexts. Nonlinear layers then execute entirely in this compact representation, where every live BSGS power is a small slice tile rather than a full ciphertext, making it $32\text{--}128\times$ smaller at our selected tile widths.

Level-adaptive, L2-aware slice tiling: A GL ciphertext is n independent RLWE slices, so unlike a CKKS ciphertext it need not be processed as one monolithic object. Terrazzo schedules every kernel over a tile of slices and treats the tile width as a per-level tuning knob, picked per level from an explicit model of L2 capacity and SM occupancy (Section 3.2). The same knob is a memory lever of a different kind: the scratch buffers of full-ciphertext operations are sized per tile rather than per full ciphertext, and our lazy BSGS evaluator uses it to run the deep polynomial evaluations of bootstrapping and nonlinear layers tile by tile.

Adaptive modulus management with grafting-free bootstrapping procedure: We adopt grafting [6], a modulus management strategy that decouples scale factors from the RNS moduli and keeps machine words fully utilized. Unlike prior deployments, we confine grafting to the application levels: bootstrapping itself runs a sprout-free procedure, so the deepest and most memory-critical circuit in the system pays no sprout overhead.

Expansion-free plaintext: In CKKS, plaintexts are encoded in a pre-initialization step: the diagonal plaintexts of the StC and CtS transforms, or the model weights for inference. This causes a large memory overhead, since each encoded plaintext is roughly an $L\times$ expansion of its cleartext, where L is the multiplicative level it is encoded at; recent work resorts to compressing the stored plaintexts to contain it [46]. Terrazzo eliminates the expansion instead of compressing it: each weight is stored as a single packed-FP32 IFFT tile that is exactly cleartext-sized and expanded in registers inside the consuming kernel (Section 3.5).

Furthermore, we employ additional memory levers to reduce resident and scratch memory and fit larger applications. Since big evaluation keys are very large in GL, we do not keep their pseudorandom a -components resident in GPU DRAM: they are regenerated from a seed with the ChaCha20 CSPRNG stream cipher, produced in registers and consumed immediately inside the consuming kernel. We also reformulate the W-axis linear transform of [47], reducing its plaintexts from diagonally encoded ring elements to single RNS residues, yielding a $256\times$ memory reduction with our parameters.

At full 256-input occupancy, Terrazzo achieves a 7.04s amortized time per input on a consumer 32 GB RTX 5090, with the complete pipeline peaking at 29.8 GB. Its A100 time is 29.98s amortized, a $2.20\text{--}20.09\times$ speedup over prior single-A100 encrypted BERT-base inference. Per-slice bootstrap speedup is $1.12\text{--}2.03\times$ on the RTX 5090 and $1.08\text{--}1.36\times$ on the A100.

Roadmap: Section 2 reviews the CKKS and GL schemes, GL bootstrapping, the GPU execution model, and our threat model. Section 3 presents our memory-aware GPU architecture: the mapping of GL primitives, slice-tiled execution, adaptive modulus management with grafting, the Y-decoded bootstrapping workflow, and the expansion-free plaintext pipeline. Section 4 details the evaluation protocol and cryptographic pa-

rameters, reports microbenchmarks and end-to-end BERT-base results, and compares against SOTA CKKS systems. Section 5 compares our framework with prior work, and Section 6 concludes the paper.

2 Background

2.1 CKKS and GL Schemes

CKKS. CKKS [7] encrypts a vector of $N/2$ complex values (*slots*) into a single element of the ring $\mathbb{Z}_Q[X]/(X^N + 1)$, where the ring degree N is a power of two (typically 2^{16}). Messages are multiplied by a scale Δ and rounded during encoding; homomorphic additions and multiplications then act slot-wise on the underlying vector. A ciphertext is a pair (b, a) satisfying $b + a \cdot s = m + e$ for the secret key s and a small noise e . To avoid big-integer arithmetic, Q is chosen as a product of word-sized RNS primes, so each ring element is stored as L residue polynomials, or *limbs*, which in effect form an $L \times N$ matrix of machine words [9]. Every multiplication squares the scale; *rescaling* divides by approximately Δ and drops one limb, so a ciphertext supports a bounded number of multiplicative *levels* before it must be refreshed by bootstrapping. Finally, multiplications and rotations change the decryption key and are followed by a *key switch* using a public evaluation key (evk) of *dnum* digit pairs over an extended modulus PQ [23]; key switching dominates the runtime of nontrivial CKKS workloads [10].

Key switching. Two kernel families dominate the cost of key switching. The *NTT* (number-theoretic transform), the finite-field analogue of the FFT, converts a limb between coefficient form and evaluation form; polynomial products are cheap (element-wise) only in evaluation form, so ciphertexts are kept there and transformed back only when a step genuinely needs coefficients. BConv (basis conversion) re-expresses a polynomial given modulo one set of RNS primes modulo another: viewing the polynomial as an $L \times N$ matrix, BConv multiplies it by a small constant conversion table, and is valid only in coefficient form. The hybrid key switch [23] composes the two. The input polynomial is cut into *dnum* *digits* (groups of limbs); ModUp lifts each digit to the extended basis PQ (an inverse NTT, a BConv, and a forward NTT per digit); KeyMult multiplies each lifted digit with its pair of evk polynomials and accumulates; and ModDown divides the accumulated pair by P (another BConv) to land back on Q , shrinking the noise the evk introduced. NTT and BConv dominate the arithmetic; reading the evk, which contains $2 \cdot \text{dnum}$ polynomials over the extended basis and is by far the largest operand, dominates the memory traffic.

No native matrix multiplication. A product against an $n \times n$ plaintext matrix (the shape GL supports natively) must be assembled from diagonal-packed plaintext multiplications and $\Theta(n)$ slot rotations ($\Theta(\sqrt{n})$ with baby-step/giant-step grouping), each rotation carrying its own evaluation key and key switch [22, 26]. Each diagonal, once encoded, is a full ring element of $L \cdot N$ machine words, so plaintext model weights inflate by orders of magnitude when prepared for encrypted inference.

The GL scheme. GL [18] is a CKKS-style approximate FHE scheme whose plaintext space is the three-variable ring

$$\mathcal{R}' = \mathbb{Z}[i][X, Y, W]/\langle X^n - i, Y^n - i, \Phi_p(W) \rangle,$$

with n a power of two and p an odd prime with generator γ ($\phi = p - 1$). Decoding is the canonical embedding τ , which evaluates a plaintext at the roots of the three defining polynomials: writing $\zeta_j = \zeta^{5^j}$ and $\eta_w = \eta^{\gamma^w}$ for primitive $4n$ -th and p -th roots of unity ζ and η ,

$$m(\zeta_j, \zeta_k, \eta_w) = M_{jk}^{(w)}, \quad j, k \in [0, n), w \in [0, \phi),$$

so one plaintext holds ϕ complex $n \times n$ matrices $\mathbf{M}^{(0)}, \dots, \mathbf{M}^{(\phi-1)}$: rows are indexed by the X axis, columns by the Y axis, and matrices (batch entries) by the W axis. Encoding is the rounded inverse map $m = \lfloor \Delta \cdot \tau^{-1}((\mathbf{M}^{(w)})_w) \rfloor$; as in CKKS, the embedding is a discrete Fourier transform, factoring into independent DFTs along the X , Y , and W axes [18, 47]. The secret key is drawn not from $\mathcal{R}'$ but from the Y -free subring

$\mathcal{R} = \mathbb{Z}[i][X, W]/\langle X^n - i, \Phi_p(W) \rangle$; a ciphertext is a pair $\text{ct}_s(m) = (b, a) \in (\mathcal{R}'_Q)^2$ with $b + a \cdot s = m + e$. All of the CKKS machinery above carries over unchanged, including scale and rescaling, RNS limbs and levels, evaluation keys and key switching, with RNS primes $q_i \equiv 1 \pmod{4np}$ so that all three axes remain NTT-friendly.

Homomorphic matrix operations. The workhorse of GL is the trace contraction: for $u, v \in \mathcal{R}'$, define

$$u \otimes v := \text{Tr}_Z(u(X, Z, W) \cdot v(Y^{-1}, Z^{-1}, W^{-1})),$$

where Tr_Z extracts the Z^0 coefficient. On encodings, $\otimes$ computes the batched matrix product $\frac{1}{n} \mathbf{A}^{(w)} (\mathbf{B}^{(w)})^H$ in every W slot [18]. Both ciphertext components are known to the evaluator, so $\otimes$ involves no homomorphic subroutine: the trace contraction reduces to ϕ independent $n \times n$ matrix multiplications over the polynomial coefficients, one per W slot, repeated for every RNS limb, which in implementation terms is one batched GEMM [18, 47]. GL supports:

- **HAdd / PAdd**: entrywise matrix addition, $(b + b', a + a')$.
- **PMult / HMult**: entrywise (Hadamard) products, which square the scale as in CKKS. The ciphertext–ciphertext form $(bb', ab' + ba', aa')$ decrypts under (s, s^2) and is relinearized by one *small* key switch (defined below).
- **Ciphertext–plaintext matmul (PCMM)**: $(b \otimes \mu, a \otimes \mu)$ for a plaintext matrix μ . Since Tr_Z is $\mathcal{R}$ -linear and $s \in \mathcal{R}$, the result decrypts under the *unchanged* key: multiplying by a plaintext matrix requires no key switch at all.
- **Ciphertext–ciphertext matmul (CCMM)**: given a second ciphertext $\text{ct}_s(\mu) = (b', a')$, the product is assembled from the four pairwise traces $(b \otimes b', a \otimes b', b \otimes a', a \otimes a')$, which decrypt under the extended key (s, s^*, ss^*) , where $s^* = s(X^{-1}, Y^{-1}, W^{-1})$ is the transposed secret key; two *big* key switches relinearize back to s [18, 47].
- **Rotations**: row rotation substitutes $X \mapsto X^{5^\delta}$ and inter-matrix rotation substitutes $W \mapsto W^{\gamma^\delta}$, each followed by a small key switch. *Column rotation* substitutes $Y \mapsto Y^{5^\delta}$, which leaves the Y -free key untouched and therefore costs no key switch.
- **Transposition**: swapping (X, Y) transposes every matrix (the key becomes $s(Y, W)$); the conjugate-inverse substitution yields the conjugate transpose (the key becomes s^*). Both are followed by a big key switch.

These key switches are not all alike, and later sections lean on the difference. In a *small* key switch the outgoing key is still an element of the Y -free base ring $\mathcal{R}$: s^2 after a Hadamard product, or a rotated key such as $s(X^{5^\delta}, W)$. Its evaluation key is then itself a base-ring object consisting of **dnum** digit pairs over $\mathcal{R}_{PQ}$, and the *same* key material serves every Y slice of the ciphertext. In a *big* key switch the outgoing key depends on Y : the transposed keys $s(Y, W)$ and s^* , the matmul key ss^* , and, in bootstrapping, keys over the auxiliary ring. Such an evaluation key must span the full ring $\mathcal{R}'$, making it up to $n \times$ larger at the same modulus. Small key switches are frequent but cheap and shared across slices; big key switches are rare but carry the largest objects in the scheme.

One ciphertext, n independent RLWE slices. Because the secret key contains no Y , collecting coefficients of like powers of Y splits a GL ciphertext into n mutually independent RLWE ciphertexts over the base ring $\mathcal{R}$:

$$\text{ct}^{(j)} = ([Y^j] b, [Y^j] a), \quad j \in [0, n),$$

where $[Y^j]$ extracts the coefficient of Y^j , and every $\text{ct}^{(j)}$ is encrypted under the same key s [18]. Security therefore reduces to standard RLWE over $\mathbb{Z}[X]/\Phi_{4np}(X)$ [18, 36], of degree $2n\phi$, equal to 2^{16} at the parameters used throughout this paper ($n = 128, p = 257$), the same ring dimension used by production CKKS deployments. We call $\text{ct}^{(j)}$ the j -th Y slice of the ciphertext. The same independence holds in the evaluation domain: after the Y -axis NTT (the form in which our ciphertexts live on the GPU), each of the n lanes is

again an RLWE ciphertext under s , and every entrywise operation, X - or W -axis transform, and key switch acts on each lane separately.

This architectural independence is the core algebraic property that enables our framework. Any κ of the n slices form a valid ciphertext fragment that can be computed on alone, at a proportional κ/n fraction of the cost and the working memory of a full-ciphertext operation. This is a degree of scheduling freedom with no CKKS analogue, which our slice-tiled execution model (Section 3.2) turns into a joint latency and memory lever. Note that the independence is a property of *ciphertexts*, not of *messages*: the encoding’s Y -axis DFT spreads every column of every message matrix across all n slices, so a single slice does not by itself carry a column of $\mathbf{M}^{(w)}$. Making message semantics local to a slice is precisely the purpose of the Y -decoded representation of Section 3.4.

2.2 GL Bootstrapping

Rescaling spends modulus, so after its last multiplication a ciphertext sits at the base modulus q_0 and cannot be multiplied further; bootstrapping refreshes it. GL adopts the four-step CKKS blueprint [11, 47]. *ModRaise* simply reinterprets the exhausted ciphertext under the full modulus Q_L . This costs nothing, but the message silently becomes $m + q_0 I$, where I is an unknown integer polynomial with small coefficients: the multiples of q_0 that modular arithmetic used to hide are now real. Removing $q_0 I$ means reducing every *coefficient* modulo q_0 , but homomorphic arithmetic acts on *slots*. The remaining steps repair exactly this mismatch: CtS (coefficients-to-slots) applies the inverse DFT homomorphically, so the coefficients appear in the slots; *EvalMod* evaluates a polynomial approximation of reduction modulo q_0 (a scaled sine) on all slots at once; and StC (slots-to-coefficients) applies the DFT to put everything back. The result decrypts to approximately m again, with fresh levels to spend.

In CKKS, CtS and StC are the expensive steps: each is a large homomorphic linear transform assembled from rotation ladders, whose key-switch count grows with the slot count and the chosen depth. GL’s encoding makes the same transforms structurally simple: encoding is an independent DFT per axis (Section 2.1), and a DFT along an axis is just a matrix multiplication, so CtS and StC collapse into a handful of PCMMs [11]. One alignment problem remains. The native PCMM is a *right* multiplication: it contracts the Y axis, while the DFTs that bootstrapping needs act along X and W . The X axis is reached by a transpose trick: transpose (a key-switched automorphism swapping $X \leftrightarrow Y$), right-multiply, transpose back. The W axis has no partner axis to swap with, so GL adjoins one: an auxiliary variable V with the same modulus, extending the ring to $\mathcal{R}_{\text{aux}} = \mathbb{Z}[i][X, W, V]/\langle X^n - i, \Phi_p(W), \Phi_p(V) \rangle$. V plays for W precisely the role Y plays for X : swap $W \leftrightarrow V$ into the auxiliary ring, right-multiply by the DFT plaintext along the freed axis, and swap back. In total, CtS and StC cost a *constant* number of key switches in place of CKKS’s rotation ladders [11].

Two consequences follow. First, with the linear stages cheap and shallow, the bootstrapping bottleneck moves to *EvalMod*, in both runtime and depth [11, 47]. Second, the cost that remains is concentrated in a few big key switches (Section 2.1): the transpose and $W \leftrightarrow V$ swaps use evaluation keys over the (extended) ring that are the largest single objects in the scheme, reaching multiple gigabytes at bootstrapping parameters, so storing, streaming, and reusing them is a first-order memory concern for any implementation. Finally, the four-stage cycle can be entered at any point: an implementation may equally run StC first (StC $\rightarrow$ ModRaise $\rightarrow$ CtS $\rightarrow$ EvalMod), placing the cheapest stage at the lowest modulus.

EvalMod is representative of how FHE evaluates any smooth nonlinearity, such as the exponential or reciprocal inside a Softmax: as a polynomial approximation of some degree d , usually expressed in the Chebyshev basis for numerical stability [4]. Evaluating the polynomial by Horner’s rule would take d sequential multiplications and d levels; the standard schedule is instead baby-step/giant-step (BSGS) [41]: compute $B \approx \sqrt{d}$ *baby* polynomials $T_1, \dots, T_B$ (the low-order Chebyshev polynomials of the input) and a few *giant* polynomials $T_B, T_{2B}, T_{4B}, \dots$ by repeated squaring, then assemble the polynomial as roughly d/B blocks, each a plaintext-weighted sum of the babies times one giant. This costs $\mathcal{O}(\sqrt{d})$ ciphertext multiplications at logarithmic depth. The price is memory: every baby power must remain live until the final block has consumed it, so BSGS holds $\Theta(\sqrt{d})$ ciphertexts live simultaneously, and the baby count B is a dial trading working memory against extra sequential multiplications.

2.3 GPU Execution Model

A GPU program is a sequence of *kernels*, each launching many *thread blocks* onto the GPU’s streaming multiprocessors (SMs). Threads hold operands in registers, and the threads of one block can cooperate through *shared memory*, a small per-SM scratchpad (about 100 KB); blocks, however, cannot synchronize with one another, so any value that crosses a kernel boundary travels through *global memory*: the L2 cache and, behind it, GPU DRAM. GPUs natively compute on 32-bit integers because 64-bit arithmetic is emulated at several times the cost, which is why the fastest FHE libraries insist on 32-bit words [9].

Memory, not compute, governs FHE on this machine. On the RTX 5090, DRAM sustains about 1.8 TB/s and the 96 MB L2 cache about 6 TB/s, only 2–4× more [10]. Outside the compute-dense NTT, FHE kernels are element-wise passes performing one or two operations per word, so their runtime is simply their traffic divided by bandwidth. The standard countermeasure, *kernel fusion*, merges adjacent element-wise steps into one kernel so that intermediates never leave registers; it is limited by mismatched parallelization patterns (the row-parallel NTT cannot merge with the column-parallel BConv) and by register pressure [10].

2.4 CKKS on GPUs and the Memory Wall

Two recent systems define the state of the art for CKKS on GPUs and serve as our baselines. Cheddar [9] makes FHE execution fully 32-bit through an RNS construction of 25–30 bit primes, signed Montgomery reduction, and extensive kernel fusion, and bootstraps 32,768 complex values in 22.1 ms on an RTX 5090, 2.18–4.45× faster than prior GPU libraries.

Its follow-up, Theodosian [10], shows where such a library actually runs: the L2 cache is now large enough to hold several ciphertexts yet too small for a whole key-switching sequence, and its bandwidth exceeds DRAM’s by only 2–4×. At the kernel level it contributes a reduction-free BConv: BConv accumulates products against a constant conversion table T , and with primes near 2^{31} the running sum overflows a 64-bit register, forcing costly intermediate modular reductions; Theodosian instead selects the moduli and table so that every complete row sum $\sum_j T[i][j] \cdot Q_j$ provably stays below 2^{64} , deleting every intermediate reduction while keeping word-sized primes. By sizing each kernel’s polynomial batch to L2 capacity (L2-aware batching) and overlapping stages with complementary bottlenecks, Theodosian reaches 15.2 ms, or 12.8 ms with further algorithmic optimizations, within 2× of the 8.8 ms bandwidth floor it derives for its parameters. The lesson: at 44–53 GB of global memory traffic per bootstrap, GPU CKKS is pressed against the memory wall itself. Terrazzo adopts both foundations at the arithmetic level, namely Cheddar’s 32-bit signed Montgomery representation and Theodosian’s overflow-free BConv accumulation, re-certified for its own sub- 2^{31} prime basis (Section 3.1).

Both systems schedule, cache, and batch in whole-ciphertext units, which are the only granularity CKKS admits.

2.5 Threat Model

We use the standard outsourced-inference model. The client holds the private input and secret key, while a semi-honest server holds the plaintext model. The client sends a GL encryption and public evaluation keys; the server homomorphically evaluates the model and returns a result decryptable only by the client. Under RLWE [36], our parameters target 128-bit security following standard guidelines [3]. Because exposing approximate decrypted outputs can leak key information [30], results remain with the key holder; deployments that publish them should use noise flooding [31]. Side channels, malicious-server verification, and model extraction through repeated queries are out of scope.

3 Our Memory-Aware GPU Architecture

An encrypted transformer inference cycles through a few stages: linear layers multiply matrices, nonlinear layers evaluate polynomial approximations, bootstrapping refreshes exhausted ciphertexts, and every stage

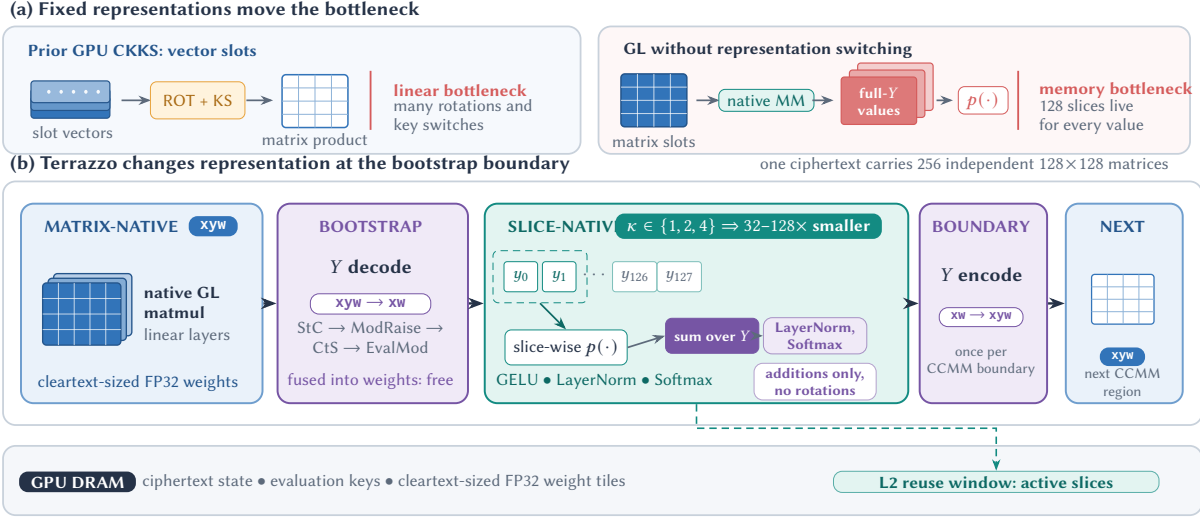


Fig. 1. Terrazzo’s representation-alternating execution. (a) A fixed representation moves the bottleneck instead of removing it: CKKS assembles every matrix product from rotations and key switches, while a GL port that never leaves the matrix encoding pays for nonlinear layers on full 128-slice values. (b) Terrazzo switches representation where it is free: the bootstrap exits into the Y-decoded xw form, with the decode folded into the plaintext operators at its boundary, so GELU, LayerNorm, and Softmax stream $\kappa \in \{1, 2, 4\}$ slice tiles with $32\text{--}128\times$ smaller intermediates and sums over Y use additions only, with no rotations; one Y encode, fused into the next layer’s weights, restores the matrix-native xyw form that CCMM requires. Below, the memory system view: the active slice tile stays L2-resident, while DRAM holds only ciphertext state, evaluation keys, and cleartext-sized weights.

consumes encoded plaintexts and evaluation keys. In GL all of them act on one object: a ciphertext whose coefficients form an $n \times n \times \phi$ block indexed by (X, Y, W) , where X and Y are the matrix dimensions and W is the batch dimension, with $\phi = 256$ matrices riding through every operation at once. This layout makes the linear stage native: a matrix product is one batched GEMM per RNS limb (Section 3.1), with no rotations to assemble. But it also makes every object enormous: 67 MB per limb per ciphertext and gigabytes at bootstrapping moduli. Consequently, a GPU port that treats these objects the way CKKS libraries must, as indivisible wholes prepared in advance, needs terabytes of VRAM.

Each stage instead gets a solution shaped to it. Executing kernels over whole ciphertexts wastes scratch and parallelism; we schedule every kernel over κ of the ciphertext’s n independent slices, with κ chosen per level against an L2 and occupancy model (Section 3.2). A nonlinear layer’s BSGS schedule holds some 30 GB of full ciphertexts live; our BERT schedule instead fuses Y -axis decoding into adjacent weight or padding plaintexts and evaluates the nonlinear layer in xw , shrinking every polynomial intermediate to a κ -slice tile, $n/\kappa = 32\text{--}128\times$ smaller (Section 3.4). Here, “sum over Y ” is the Softmax row sum of exponentials or the LayerNorm feature sum used for each token’s mean and variance; GELU needs no sum. Decoding the Y -axis into xw exposes each column as an independent coefficient slice, so these sums are direct slice additions rather than rotations and key switches. The result occupies one slice while retaining every X row and W batch lane. Keeping 32-bit words full requires grafting, but a graft sprout participates in every computation it touches; we confine it to the application levels, and bootstrapping, the deepest circuit, runs sprout-free (Section 3.3). Pre-encoded plaintexts expand $L\times$ over their cleartext and repeat the same operator in all ϕ batch slots; we store weights cleartext-sized and encode them in registers inside the consuming kernel (Section 3.5). And big evaluation keys, which occupy several gigabytes each, are not stored at all in their pseudorandom half, which is regenerated from a seed on the fly inside the consuming keyswitch kernels.

Figure 1 previews the resulting execution. Together these measures bring the working set from terabytes to a peak of 29.8 GB, on which the entire pipeline, including bootstrapping and attention, executes.

3.1 Mapping GL Primitives to the GPU

The u, v representation. GL’s rings are built over the Gaussian integers $\mathbb{Z}[i]$, but no complex arithmetic ever reaches the GPU. Because every $q_i \equiv 1 \pmod{4np}$ satisfies $q_i \equiv 1 \pmod{4}$, the imaginary unit has a residue ω modulo q_i and $\mathbb{Z}[i]/q_i$ splits by CRT into two conjugate components: the coefficient $a + bi$ becomes the two independent residues $u = a + \omega b$ and $v = a - \omega b$ [11, 18]. We map this split directly onto the memory system: each limb is two ordinary $\mathbb{Z}_{q_i}$ planes on which every ring operation acts componentwise, and complex conjugation, which GL uses constantly (conjugate transposes, attention’s paired real/imaginary lanes), is simply an exchange of the u and v planes. All residues are signed 32-bit words in centered Montgomery form [9], and reduction is scheduled lazily against a per-parameter bound ledger. Basis conversion follows the same discipline through Theodosian’s reduction-free BConv (Section 2.4): our parameter search certifies, for every conversion row of the selected sub- 2^{31} basis, that the complete inner product fits a single signed 64-bit accumulator, so no intermediate reduction ever runs.

Two easy transforms and one hard one. The evaluation encoding factors per axis (Section 2.1), and the axes are not equal. X and Y carry twisted cyclic power-of-two NTTs, implemented as standard butterflies with the $X^n = i$ twist folded into the tables (u and v differ only in the sign of the twist). The W axis is the prime cyclotomic $\Phi_p(W)$: no power-of-two structure exists, and the transform is evaluated by Rader’s reduction: reindex by the generator γ , turning the p -point DFT into a cyclic convolution of length $\phi = p - 1 = 256$, itself computed as a forward 256-point NTT, a pointwise product with a fixed kernel spectrum, and an inverse 256-point NTT, plus zero-frequency corrections. One W pass therefore hides three transforms: about 9 modular multiplications per coefficient against 3.5 for X . Moreover, W is the hot axis, crossed by every key switch.

How the kernels absorb it. The design leans on the memory-wall arithmetic of Section 2.3: at ~ 1.8 TB/s, the GPU sustains roughly eight integer multiplies per byte it moves, and even Rader’s inflated count comes to ~ 3.4 operations per byte, so every axis kernel is DRAM-bound, and the scarce resource is passes over the data, never multiplies. Each axis is therefore one kernel making exactly one read and one write per plane, with butterflies in registers around a shared-memory tile, and consecutive axis passes hand their data through L2. The layout does the rest. We keep two coefficient orders and choose per phase: the matrix-native $[w][x][y]$, in which every W slot is one contiguous matrix ready for the GEMM consumers, and the slice-major $[y][w][x]$ used by the slice-tiled phases of Section 3.2, including small key switches, slice-tiled multiplication, and EvalMod, in which a κ -slice tile is one contiguous object. In either order each transform keeps a contiguous batch axis, so Rader’s generator permutations degrade into renames of whole coalesced rows and the W kernels run without a single divergent branch. Finally, the transform bodies expose load/store fusion hooks, so adjacent pointwise work, including basis-conversion ingest, key-switch accumulation, and the scalar diagonals of Section 3.4, rides the same pass instead of buying its own. Unlike CKKS, GL’s per-axis twiddle tables are only a few kilobytes and remain cache-resident, so Terrazzo precomputes them rather than generating them inside kernels [9].

On these foundations the scheme’s primitives map directly: PCMM is a batched per-(limb, part) GEMM over the ϕ W slots, and the hybrid key switch’s ModUp, KeyMult, and ModDown stages become batched basis-conversion GEMMs and multiply-accumulate passes over the same planes. The bootstrap’s X -GEMM batches every RNS limb, u, v part, and ciphertext component as an independent plane. Each CUDA block stages the shared 128×128 transform matrix in shared memory, while warps hold ciphertext rows in registers and use shuffles with chunked modular accumulation to compute the output in place. The same principle halves the largest operand of a *big* key switch, the multi-gigabyte keys that bootstrapping and CCMM require: an evaluation-key digit’s a component is uniformly random, so for these keys we store only b and let the KeyMult kernel regenerate a from a ChaCha20 seed in registers as it accumulates, trading arithmetic for bandwidth in a kernel bound by the latter.

3.2 Slice-Tiled Execution

The tile. Because any κ of a ciphertext’s n Y slices form a valid fragment (Section 2.1), the unit our scheduler hands a kernel is not a ciphertext but a κ -slice tile. Tiling serves two masters at once. The first is memory: a nonlinear stage’s BSGS schedule holds $\Theta(\sqrt{d})$ intermediates live, and at κ slices each is n/κ times smaller than a full ciphertext. This is what turns the 30 GB scratch of Section 3.4 into hundreds of megabytes. The second, less obviously, is latency: the fastest κ is neither 1 nor n , and choosing it per level is worth tens of percent.

The kernel fusion of Section 3.1 merges passes that *share an operand* into one fatter thread block: a fused ciphertext–plaintext kernel reads the plaintext plane once and serves both ciphertext components; a key-switch accumulation reads each evaluation-key digit once and serves every slice in the tile. Each such fusion halves its traffic, but also halves the number of independent blocks a launch spawns. At $\kappa = 1$ the fused kernels no longer fill the GPU’s SMs, and every launch’s fixed cost is amortized over the least possible work. Widening to $\kappa = 2$ restores the grid and lets one read of the evaluation key, the largest operand in any key switch, serve two slices, so $\kappa = 2$ is faster *per slice* than $\kappa = 1$ at every level we measured, and it is our general default for nonlinear work.

Deeper in the application, wider tiles. A launch’s parallelism scales with (active limbs) $\times$ (two u, v parts) $\times$ (κ slices). Rescaling erodes the first factor: as the modulus chain shortens, each slice simply offers fewer planes, and occupancy sags again. The scheduler compensates by widening the tile at low levels. Our GELU, evaluated near the bottom of the application chain, runs at $\kappa = 4$, where the same configuration that is counterproductive at bootstrap depth is over 40% faster than $\kappa = 1$.

The L2 ceiling. Raising κ blindly fails in the other direction. The tile’s working set, consisting of κ slices across all active limbs plus the shared evaluation-key digits, must flow through the L2 cache (96 MB on the RTX 5090, for example) for the reuse that motivated the tile to exist at all; past that footprint, the fatter tile evicts exactly the operands it meant to share, and per-slice latency rises despite identical arithmetic. We measure this crossover directly: at bootstrap-depth chains, $\kappa \in \{4, 8\}$ is slower per slice than $\kappa = 2$. The scheduler therefore selects κ per level against this two-sided model, with occupancy pushing from below and L2 capacity pressing from above, rather than fixing it globally. Furthermore, Figure 1’s reuse window, which contains the active slices, is precisely the set this choice keeps L2-resident.

3.3 Grafting-Based Modulus Management

Incompatibility with small-prime systems. Fast GPU FHE keeps every residue in a native 32-bit word (Section 2.3); Cheddar and Theodosian achieve this with the 25–30 prime system, drawing RNS primes between 2^{25} and 2^{30} and managing scales with rational rescaling [9]. GL cannot simply import that construction, because its prime population is far more constrained: every prime must keep all three axes NTT-friendly, $q_i \equiv 1 \pmod{4np}$ (Section 2.1), and at our $n = 128$, $p = 257$ this means $q_i \equiv 1 \pmod{131,584}$. Counting the primes that satisfy it: there exist only 13 such 25-bit primes, 30 26-bit primes, and 53 27-bit primes, while one bootstrappable GL chain already consumes 59 (our 41 Q primes, 15 P primes, and 3 sprout shadows, Section 4). Small-prime systems are therefore arithmetically impossible for GL, entirely apart from their wasted word bits. The population only becomes comfortable at the word ceiling, with 415 primes at 30 bits and 772 at 31, so GL is pushed to a uniform basis of near-word-width primes below 2^{31} . That choice keeps words full but creates the opposite problem: the application scale $\Delta = 2^{35}$ now fits no single prime and wastes half of two, so no sequence of whole-prime rescales can realize the level widths the application needs.

Grafting decouples scale from primes. Grafting [6] resolves exactly this incommensurability by placing a power-of-two *sprout* inside the modulus [6]: at level ℓ the ciphertext modulus is

$$Q_\ell = 2^{\alpha_\ell} \prod_{i \leq \ell} q_i, \quad 0 \leq \alpha_\ell \leq \alpha_{\max},$$

and a rescale drops whole primes while shifting the sprout by a signed amount, dividing by exactly $2^{\alpha_\ell - \alpha_{\ell-1}} \prod_{\text{dropped}} q_i$. The sprout supplies the fractional bits the prime menu cannot: a 35-bit application level is realized as one ≈ 30 -bit prime plus a ≈ 5 -bit sprout shift, and because powers of two divide exactly, scale bookkeeping stays exact where an approximate prime-only divisor would drift. We plan the entire ladder jointly, including every level’s prime drops and sprout shifts across all 41 Q primes, so that each stage boundary lands on its planned scale and α wanders only within $[0, \alpha_{\max}]$.

Transparent sprout representation. Cheddar considered grafting and rejected it for GPUs: transforming a power-of-two limb such as 2^{30} requires embedding it into a ring with modulus above $N \cdot 2^{60}$, forcing 128-bit arithmetic [9]. Our implementation never transforms a power-of-two limb at all. The sprout’s content rides as *shadow limbs*: residues of its centered lift modulo a few ordinary NTT-friendly word-sized primes s_j . The sprout bound dictates how many. A CRT fold over the shadow basis recovers a centered value exactly whenever the value’s width stays below $\log_2(\prod_j s_j) - 1$, and the widest value the pipeline folds is a product of two sprout-bounded operands plus accumulation slack, about $2\alpha_{\max} + \lambda$ bits. Thus, with $s_j < 2^{31}$, $\lceil (2\alpha_{\max} + \lambda)/31 \rceil$ shadow primes suffice; our chains carry three. And because 2^α divides $2^{\alpha_{\max}}$, residues taken at $\alpha_{\max}$ serve every sprout width at once, so no per- α material exists. Every bulk kernel, whether NTT, GEMM, key-switch, or Hadamard, processes shadows as three additional prime limbs, with the same 32-bit code paths and no divergence; the exact $\mathbb{Z}_{2^\alpha}$ residue is materialized only inside a planned rescale, by the shadow CRT fold followed by a mask and shift, coefficient-wise and cheap. Plaintext operators extend to shadows by $\mathbb{Z}$ -linearity, through the same expansion-free pipeline as their prime residues (Section 3.5). To our knowledge this is the first GPU grafting realization.

The bootstrap runs sprout-free. The sprout is not free: every object it touches carries its shadow limbs, and every planned rescale pays a fold epilogue, which is a per-operation cost. But the sprout is only *needed* where a stage’s width is incommensurate with the prime menu, and while the application scale is fixed by the model’s precision needs, the bootstrap’s stage widths are ours to choose. The two cases differ fundamentally. A 35-bit level admits no whole-prime realization at all: it would take primes around 17–18 bits, and *no* prime satisfying the $4np$ congruence exists below 21 bits. The widths of q_0 , StC, CtS, and the EvalMod levels, by contrast, are free parameters, and we tune each to an exact multiple of whole prime widths of 58 or 60 bits, two whole primes each. This costs nothing, because it is what those stages want anyway: CtS and EvalMod need a high working scale for transform and polynomial precision, and a wide base modulus q_0 buys bootstrap accuracy. Every bootstrap rescale then becomes a fused multi-prime drop with no sprout motion, and the planner’s contract reduces to two boundary conditions: $\alpha = 0$ on entering StC, and whatever α the next application level requires on exiting EvalMod. Everything in between, the deepest and most operation-dense circuit in the pipeline, runs with no sprout, no shadow limbs, and no fold epilogues. Grafting is confined to the shallow application levels, precisely the region where its fractional bits earn their keep.

3.4 A Modified GL Bootstrapping Workflow

The representation problem. Standard EvalMod-based GL bootstrapping [11] refreshes a ciphertext in the full three-axis encoding **xyw**: the output carries the DFT along X , Y , and W , which is what CCMM consumes; PCMM is representation-agnostic, since any change of basis folds into its public weights. The price is paid by every *polynomial* layer. Under the Y -axis DFT, a Y slice does not hold a column of any message matrix $\mathbf{M}^{(w)}$ but a column of its Y transform (Section 2.1), so a nonlinearity cannot be evaluated slice by slice: EvalMod, Softmax, GELU, and LayerNorm must treat the whole $n^2\phi$ object as one vector of values, and their BSGS schedules (Section 2.2) hold $\Theta(\sqrt{d})$ such objects live at once. At application moduli, this means roughly 16 full ciphertexts and some 30 GB of scratch for a single GELU.

Bootstrapping in the Y -decoded representation. Our pipeline therefore runs the bootstrap and every nonlinear layer in the Y -decoded representation **xw**: X and W keep their slot transforms while the Y axis is returned to coefficient form. This one change makes message semantics local. With Y decoded, the j -th Y slice is an ordinary RLWE ciphertext whose slots hold the j -th columns of all ϕ message matrices, so a polynomial

acts on each slice independently, and the slice-tiled execution of Section 3.2 applies to nonlinear layers at $\kappa \in \{1, 2, 4\}$: every BSGS intermediate shrinks by $n/\kappa = 32\text{--}128\times$. The refresh cycle is entered at StC, the cheapest stage at the lowest modulus: StC applies the DFT along W and X only (its two stages sharing one rescale), ModRaise lifts the result, CtS undoes both transforms at the raised modulus, and EvalMod, the dominant stage, runs entirely on $\kappa = 2$ slices. The bootstrap consumes and produces $\mathbf{xw}$; it never carries a Y -transform plaintext, an extra modulus limb for its plaintext scale, or a Y -related key. The same locality extends to the refresh itself: an auxiliary value that occupies a single slice, such as a LayerNorm variance or a Softmax row norm, is bootstrapped as a $\kappa = 1$ fragment, with CtS and EvalMod acting on that one slice, so the pipeline’s frequent scalar refreshes cost a small fraction of a full-ciphertext bootstrap.

Zero-cost Y transforms via weight fusion. The Y decode and encode are fixed $n \times n$ linear maps acting on the Y axis, the same axis a PCMM weight matrix multiplies. They therefore compose with adjacent weight matrices *in the clear*, before encoding, exactly as the expansion-free pipeline of Section 3.5 prescribes: replacing a weight $\mathbf{W}$ by its product with the Y -DFT matrix or its inverse costs no level, no homomorphic operation, and no additional plaintext object.

This is best seen at the level of the coefficients the GPU actually multiplies. Per W slot, PCMM is always the same operation: the ciphertext’s $n \times n$ coefficient matrix $\mathbf{C}$, with Y in the coefficient basis the GEMM contracts (Section 3.1), is right-multiplied by a public matrix $\mathbf{B}$. What the representation changes is only what $\mathbf{C}$ ’s columns *mean*: with $\mathbf{F}$ the Y -axis DFT matrix, a matrix-native ciphertext carries the message $\mathbf{M} = \mathbf{C}\mathbf{F}$, while a Y -decoded one carries $\mathbf{M} = \mathbf{C}$ directly. To realize the message-level product $\mathbf{M} \mapsto \mathbf{M}\mathbf{A}$ between *any* two representations, it suffices to encode

$$\mathbf{B} = \mathbf{F}^{\delta_{\text{in}}} \mathbf{A} \mathbf{F}^{-\delta_{\text{out}}}, \quad \delta = 1 \text{ for } \mathbf{xyw}, \quad \delta = 0 \text{ for } \mathbf{xw},$$

a product computed in the clear on the public matrix. The ciphertext side never changes, so one PCMM kernel can preserve either representation or convert in either direction, all at identical cost. The $\mathbf{xw}$ -to- $\mathbf{xw}$ case is worth pausing on: for a Y -decoded ciphertext both corrections vanish: $\mathbf{B} = \mathbf{A}$, the cleartext weight itself. Thus, a decoded ciphertext takes a full matrix product and *stays* decoded, with no transform bracket around it; our residual and LayerNorm-scaling products run exactly this way. Our BERT schedule fuses every Y encode and decode into an adjacent weight or score-padding plaintext. Thus, no separate Y -transform plaintext or homomorphic Y transform is needed. CCMM is the one operation that genuinely requires the matrix-native form: both of its operands are encrypted, so neither can absorb a change of basis into a public matrix, and the trace contraction reads the Y axis in its encoded form.

The W transform: BSGS over rotations with scalar diagonals. The remaining design choice is the W -axis DFT inside StC and CtS. The original workflow [11] reaches the W axis through the auxiliary ring: batch ciphertexts into the variable V , swap $W \leftrightarrow V$, right-multiply, and swap back (Section 2.2). This costs a constant number of key switches, but its memory shape is wrong whenever $n < \phi$. An auxiliary-ring element spans $n\phi^2$ coefficients against a ciphertext’s $n^2\phi$, a factor of ϕ/n , i.e. *two* GL ciphertexts at our $n = 128$, $\phi = 256$. Every W stage then materializes double-width intermediates; a single-ciphertext refresh must still allocate and carry a dead second bank; filling that bank productively requires pairing bootstraps, which makes the transform’s efficiency application-dependent; and the swaps consume big-ring evaluation keys, the largest objects in the scheme. We instead evaluate the W -DFT directly, adapting the BSGS W -axis inversion of [47]. Any linear map along the W axis can be written in diagonal form, as a sum of W rotations each multiplied by a diagonal plaintext. For the W -DFT the diagonals collapse: reindexing coefficients and slots by the generator γ (Rader’s permutation) turns the DFT on the ϕ points η^{γ^w} into a cyclic convolution, and the diagonals of a convolution are *constant*: $\text{DFT}_W = \sum_{t=0}^{\phi-1} c_t \rho_t$, where ρ_t is the native inter-matrix rotation $W \mapsto W^{\gamma^t}$ and each c_t is a *single* scalar root of unity: the diagonal plaintexts collapse to *scalar diagonals*, using one value where the diagonal method carries ϕ (Section 3.5). Each term is one small key switch with base-ring keys shared across all Y slices, and the scalar diagonal rides the key-switch inner product as a single factor: no plaintext multiplication, no auxiliary ring, and no intermediate wider than one ciphertext, at every batch size. Even the usual BSGS correction of counter-rotating each diagonal plaintext into its block is vacuous here because a scalar is invariant under rotation.

CtS must run the exact inverse. Correctness ties the two linear legs together: ModRaise sits between them, so whatever transform StC applied to reach coefficient form, CtS must apply the exact inverse of it; otherwise, the values EvalMod reduces are not the coefficients StC produced, and the refreshed message decodes through the wrong basis. CtS therefore runs the two inverse stages in reverse order (X first, then W), and its W stage is the *same* rotation schedule executed with the inverse scalar diagonals: the conjugate roots scaled by $1/p$, with a constant correction that folds the prime cyclotomic’s missing zero frequency into the same ϕ scalars. Forward and inverse legs share rotation keys, schedule, and code; only the scalar-diagonal tables differ. Because CtS runs at the raised modulus, where every object is $\sim L \times$ larger, it is CtS that sets the W stage’s memory behavior, making the BSGS schedule itself the natural memory dial.

Choosing B . With baby-step count B , the sum is evaluated as $\lceil \phi/B \rceil$ blocks of B rotations: the babies are materialized once from a hoisted decomposition shared within each block, and one giant rotation lands each block’s contribution, for $B + \phi/B$ key switches in place of ϕ . The same B is also a memory dial: a wider block means a wider hoisted decomposition and wider extended-basis accumulators, and these live at the raised modulus, where every byte weighs most. We therefore choose B as a balance between latency optimality and memory efficiency rather than at either extreme; every B runs the same schedule with the same scalar diagonals and rotation keys, so the choice is re-tuned per parameter set by measurement.

3.5 Expansion-Free Plaintext Pipeline

Every linear stage evaluates against plaintext operators: model weights and biases through PCMM, the DFT matrices of CtS and StC, and public masks. Conventionally, a plaintext is prepared exactly like a ciphertext component: encode, then store one limb of $n^2\phi$ residues per RNS prime. The cleartext behind this object is one $n \times n$ complex matrix per W slot, and in inference it is almost always the *same* matrix in every slot, because the batch axis replicates one model over ϕ independent samples. The stored form is therefore inflated twice over its content: by $\ell + 1$ from the RNS and by ϕ from the batch. At our parameters ($n = 128$, $\phi = 256$, at which we quote all sizes in this subsection) a level-5 operator occupies 201 MB for a matrix whose cleartext fits in 131 KB, and a transformer needs hundreds of such operators. This subsection removes both factors, by storing each plaintext in the least-expanded form its structure admits and completing the encoding only where, and only as far as, an operation actually requires it.

Sources of plaintext expansion. Encoding is the rounded inverse embedding followed by per-prime work (Section 2.1): (i) apply $\Delta \cdot \tau^{-1}$, a complex-valued DFT along each axis; (ii) round once, producing an integer element of $\mathcal{R}'$; (iii) reduce that element modulo each q_i ; and (iv) bring each residue limb into the form the consuming operation needs. Steps (i) and (ii) are prime-independent: they can be computed once and stored once, at cleartext size. Prime-dependence enters only at (iii), which is elementwise and cheap, and at (iv), which for a matrix operand of $\otimes$ amounts to a single W -axis NTT: the trace contraction reads the X and Y axes directly as a coefficient matrix, but it treats each W slot as an independent matrix product, so the W axis must stand in evaluation form. That transform evaluates the operator at the p -th roots of unity *of the current prime*: roots exist modulo every q_i (this is what $q_i \equiv 1 \pmod{4np}$ buys), but they differ from prime to prime. Thus, unlike everything before the rounding, the W -axis NTT cannot be hoisted out of the RNS. In general it must be re-run per limb, after that limb’s reduction.

General batches. When the ϕ matrices genuinely differ, this is the whole story: we store the payload once at cleartext size ($n^2\phi$ scalars, 34 MB, no RNS factor) and, at each use, run steps (iii) and (iv) per limb, consisting of a reduction and one W -axis NTT, into a small scratch element that the operation consumes and releases. The $(\ell + 1) \times$ storage expansion is gone; the price is one transient scratch plaintext and one W -axis NTT per limb per use.

Batch-constant operators. Almost every operator in inference is batch-constant: $\mathbf{B}^{(w)} = \mathbf{B}$ for all w . Write the operator as a polynomial in W with matrix coefficients, $\beta(W) = \sum_{d < \phi} \beta_d W^d$. Interpolating a constant function on the ϕ evaluation points has the constant polynomial as its unique solution: $\beta_0 = \mathbf{B}$ and $\beta_d = 0$

for all $d > 0$. The W -axis NTT still formally applies, but on a degree-zero polynomial it degenerates into a broadcast: for every prime q_i and slot w , $\beta(\eta_i^{\gamma_w}) = \mathbf{B}$, and hence $(\text{NTT}_W \beta)_w = \mathbf{B}$. This is independent of both the slot index and the prime’s evaluation grid. A broadcast never needs to be computed or materialized: the single $n \times n$ matrix simply stands for all ϕ slots wherever the operator is consumed. Both inflation sources attached to the W axis disappear at once: the ϕ -fold storage because all slots share one matrix, and the per-prime transform because the one step that had to run per prime now acts as the identity on the stored object. Batch-constant plaintexts take one of two forms, differing in how far the encoding is carried:

- *Residue form.* Steps (i) to (iii) run once; we keep $\ell + 1$ residue matrices of size $n \times n$, occupying 786 KB at level 5, with only the RNS factor remaining. This form is exact and reusable, and carries the few operators reused at every refresh: the X -axis DFT plaintexts of StC and CtS and the Softmax masks below.
- *Deferred form.* Encoding stops *before* the rounding: we store only the floating-point payload of step (i). For a real weight matrix, whose payload is conjugate-symmetric, this requires one real scalar per weight, or 66 KB per $n \times n$ tile. Steps (ii) and (iii) execute inside the consuming operation itself. This is sound because rounding is deterministic: every limb rounds the same payload to the same integer element and then reduces it modulo its own prime, so the residues remain residues of one integer element, exactly as a single up-front rounding would produce. Storage equals cleartext size, meaning the expansion is zero. As a further benefit, the scale binds late: Δ and any public prefactor are applied during the deferred rounding, so one stored payload serves every level, scale, and prefactor it meets. All model weights and biases in our BERT experiments travel in this form.

In our BERT schedule, every Y encode and decode is fused into adjacent weight or padding plaintexts, so no additional Y -transform plaintext is stored.

Masks. The third kind of plaintext is the public mask. Attention over a padded sequence must suppress the scores of padding tokens, and the padding pattern is public, so the mask is a plaintext object. In the attention score matrix, X indexes query tokens, Y indexes key tokens, and W indexes the packed input sequences. Consequently, a key-padding mask depends on Y and, for nonuniform sequence lengths, W , but is constant along X . We incorporate the multiplicative 0/1 selector directly into the Y -decode PCMM and apply the corresponding Y -decoded additive correction natively by PAdd in the $\mathbf{xw}$ representation. Together, these operations preserve the intended low-score contribution for padded keys without requiring a separate selector plaintext or plaintext multiplication. The fused PCMM uses the level transition already required by the Y -decode, while PAdd consumes no level, so padding does not increase the pipeline’s multiplicative depth and keeps the downstream Softmax normalization inside its calibrated range. Per-head masks are cheaper still: they are composed into the plaintext projection weights before encoding and reach the GPU at no additional online cost.

In sum, each plaintext is stored in the least-expanded form its structure admits: the 201 MB fully expanded operator shrinks to 34 MB for a general batch, 786 KB in residue form, 66 KB deferred, and 12 KB of scalar diagonals. The missing part of the encoding is either reconstructed transiently (one W -axis NTT), completed in place (a deferred rounding), or shown to be trivial (a broadcast).

4 Experimental Evaluation

We implemented Terrazzo from scratch in C++ and CUDA 13.0. All experiments were run on a server with an AMD Ryzen Threadripper 7970X CPU (32 cores), 157GB of RAM, running Ubuntu 24. Our primary evaluation GPU is an NVIDIA GeForce RTX 5090 with 32 GB of DRAM. For comparison against related works, we additionally evaluate on an NVIDIA A100 PCIe with 80 GB of DRAM. Unless stated otherwise, reported latencies are warmed steady-state measurements using CUDA events, and every experiment retains an end-to-end encrypted correctness gate: results are decrypted and compared against a cleartext reference.

Unless stated otherwise, all experiments use one 128-bit-secure parameter set [3]: the GL ring with $n = 128$ and $p = 257$ ($\phi = 256$), giving RLWE dimension 2^{16} per Y slice; RNS primes below 2^{31} , so that

Table 1. Latency of GL primitives on the RTX 5090 (warmed, CUDA events). Q limbs is the active input basis of the operation. Per-unit latency is per matrix for PCMM, CCMM, and conjugate transpose, and per Y slice otherwise.

Primitive	Q limbs	Total (ms)	Per unit (ms)
HMult + relin + rescale ($\kappa = 2$)	39	30.59	0.239
PCMM + rescale	21	69.36	0.271
CCMM + relin + rescale	21	249.58	0.975
Conjugate transpose	21	111.78	0.437
StC	4	115.92	0.906
CtS	41	572.72	4.474
EvalMod ($\kappa = 2$)	39	729.64	5.700
Full bootstrap (batch 1)	–	1458.19	11.39

Table 2. Bootstrap latency per ring- 2^{16} slice. One GL refresh covers $n = 128$ slices; each CKKS row refreshes one ciphertext of the same ring dimension. A100 is the 80 GB model. $\dagger$ marks matched-configuration measurements reported by Theodosian ($\Delta = 2^{35}$, 15 levels); other rows use native configurations. $\log Q_{\text{mult}}$ is the post-bootstrap application modulus budget, including q_0 and excluding transform-reserved limbs. Values marked $\approx$ derive from stated levels and scale without the unstated base modulus. Prec. is $-\log_2$ of the maximum reported bootstrap error [10]. Terrazzo latency is the full refresh time divided by 128.

System	GPU	$\log QP$	$\log Q_{\text{mult}}$	Prec.	Lat. (ms)
Cheddar † [9]	RTX 5090	1711	575	18.57	23.18
Cheddar [9]	A100	1776	≈ 575	–	40.0
Theodosian † [10]	RTX 5090	1711	575	18.57	16.21
Theodosian + algo. opt. †	RTX 5090	1711	575	18.70	12.75
Theodosian [10]	A100	1776	≈ 575	–	31.8
Cerium [25]	A100	1782	–	–	34.2
HyperDrive [14]	A100	<1536	576	–	38.91
Terrazzo	RTX 5090	1665	585	18.53	11.39
Terrazzo	A100	1665	585	18.53	29.41

every residue fits a signed 32-bit word, with 41 Q primes and 15 P primes ($PQ < 2^{1665}$, within the 1747-bit modulus budget at this dimension) and $\text{dnum} = 3$; scale $\Delta = 2^{35}$ with 15 application levels; and a dense ternary secret of Hamming weight 1024. The bootstrap circuit allocates 60 bits to the base modulus q_0 and 58 bits each to StC, CtS, and its nine EvalMod levels, and runs sprout-free; the graft sprout (three shadow primes, $\alpha \leq 29$) is confined to the application levels (Section 3.3).

4.1 Microbenchmarks of GL Primitives

Table 1 reports warmed latencies of the core GL primitives. The native matrix products confirm the scheme-level promise: a PCMM with rescaling costs 0.271 ms and a CCMM with relinearization and rescaling costs 0.975 ms per 128×128 matrix, with no rotations anywhere. Within the bootstrap, the two raised-modulus stages dominate: CtS and EvalMod account for 1.30 s of the 1.46 s total, while StC, running at only 4 limbs, confirms the value of entering the refresh cycle at the lowest modulus.

4.2 CKKS Bootstrap Comparison

Table 2 compares our bootstrap against the fastest reported CKKS bootstraps, normalized per ring- 2^{16} RLWE slice (Section 2). Cheddar and Theodosian bootstrap a full-slot ciphertext back to 15 multiplicative levels at scale $\Delta = 2^{35}$ [10]; Cerium refreshes to level 18 over a 28-bit RNS basis [25]. Terrazzo evaluates a degree-38 polynomial approximation of the modular reduction, against the degree-30 approximations these systems use, so its latency is not bought with a cheaper EvalMod.

Table 3. End-to-end encrypted BERT-base inference (128 tokens, single GPU). Powerformer retrains the model; all other systems use unmodified BERT-base. Baselines are single-input A100 latencies; Terrazzo amortizes one 256-input batch. Speedups use Terrazzo’s A100 latency.

System	Hardware	Latency	A100 speedup
THOR [37]	A100	602.4 s	20.09×
Powerformer [39]	A100	344.5 s	11.49×
MOAI [53]	A100	284.0 s	9.47×
Zephyr [45]	A100	107.9 s	3.60×
Tricycle [33]	A100	100.5 s	3.35×
Cerium [25]	A100	66.0 s	2.20×
Terrazzo	A100	29.98 s	—
Terrazzo	RTX 5090	7.04 s	—

At 11.39 ms per refreshed slice, Terrazzo on the RTX 5090 is $2.03\times$ faster than Cheddar and $1.42\times/1.12\times$ faster than Theodosian without/with its algorithmic optimizations, at the matched configuration ($\Delta = 2^{35}$, 15 levels), while returning a slightly larger application budget (585 vs. 575 bits) at comparable precision. At the same parameters, the A100 latency is 29.41 ms per slice, a $1.08\text{--}1.36\times$ speedup over the A100 baselines.

4.3 BERT-base Inference Comparison

Table 3 compares end-to-end encrypted BERT-base inference against prior private-inference systems. Client communication is one upload and one download: the input enters as three packed $\mathbf{xw}$ ciphertexts, and the returned result is a single 2.6 MB RLWE slice (five Q limbs) carrying the classification logits in one coefficient, making it $128\times$ smaller than a full GL ciphertext, with zero host-device ciphertext traffic in between, enforced by a transfer-counting gate in every reported run.

The batching semantics of this comparison deserve stating plainly. GL’s W axis is a native batch dimension: one encrypted forward pass always carries 256 independent inputs, and its wall-clock time is the same whether one or all 256 lanes are occupied. On the RTX 5090 the full end-to-end pass completes in 1802 s of wall-clock time, or 7.04 s amortized per input at full occupancy. The baseline rows are single-input A100 latencies, so we report the RTX 5090 result without a cross-hardware speedup. On the A100, one batch completes in 7675 s (29.98 s per input), yielding a $2.20\text{--}20.09\times$ speedup over the A100 baselines. We emphasize that all kernel launch shapes and tile configurations were optimized specifically for the RTX 5090’s memory hierarchy and 96 MB L2 cache. Even without tuning the pipeline specifically for the A100 GPU, Terrazzo still significantly outperforms all prior single-A100 baselines, leaving clear headroom for further gains through architecture-specific tuning.

Table 4 lists each operation’s representation and ciphertext flow: outputs stream one at a time, slices hold reduction auxiliaries, and partial products accumulate into three hidden-state ciphertexts. Table 5 compares the per-operation execution times against the systems that report such breakdowns. Our Softmax evaluates the exponential-based schedule of Cho et al. [8] as instantiated in the CKKS Llama-2 implementation of Park et al. [40], executed over Terrazzo’s slice-tiled evaluator: a per-layer calibrated constant subtraction places the scores in a width-96 interval below zero, the exponential is evaluated there, and three normalize-and-square rounds restore the softmax normalization. GELU is a single degree-255 polynomial fitted on $[-64, 144]$, while each LayerNorm evaluates a degree-127 inverse-square-root polynomial with one Newton correction over a layer-specific calibrated interval.

Real-World Privacy Applicability. We evaluate Terrazzo by fine-tuning BERT-base (uncased) on SST-2 and testing on its public 1,821-example split. While movie reviews are not inherently privacy-sensitive, they provide a standard, reproducible baseline for transformer inference. Crucially, because FHE evaluation follows a data-oblivious, fixed circuit, the latency, memory footprint, and throughput we report for SST-2 apply identically to any 128-token input. Therefore, these benchmark results directly extrapolate to real-world, privacy-critical NLP tasks of the same sequence length (such as classifying sensitive patient medical notes, filtering confidential financial communications, or analyzing proprietary legal contracts) without any

Table 4. Full GL schedule of one Terrazzo BERT-base encoder layer. “Repr.” is the Y -axis form; “GL CTs (in→out)” gives the full-ciphertext flow, with “str.” streamed, “acc.” accumulated, and “+ slices” additional compact slices; “Levels” is the input → output application level. Y -encode/decode is fused into adjacent PCMM operators.

Component	Operation	Repr.	GL CTs (in→out)	Levels	(Data dimensions) $\times$ (count)
Attention	$Q/K/V$ projections (PCMM, Y -encode fused)	$\mathbf{xw} \rightarrow \mathbf{xyw}$	$3 \rightarrow 36$ str.	$6 \rightarrow 5$	$(\mathbb{R}^{128 \times 768} \times \mathbb{R}^{768 \times 768}) \times 3$
	QK^T (CCMM)	$\mathbf{xyw}$	$24 \rightarrow 12$	$4 \rightarrow 3$	$(\mathbb{R}^{128 \times 64} \times \mathbb{R}^{64 \times 128}) \times 12$
	score padding (Y -decode fused PCMM + PAdd)	$\mathbf{xyw} \rightarrow \mathbf{xw}$	$12 \rightarrow 12$	$3 \rightarrow 1$	$\mathbb{R}^{128 \times 128} \times 12$
Bootstrap	full-ciphertext refresh	$\mathbf{xw}$	$12 \rightarrow 12$	$1 \rightarrow 16$	$\mathbb{R}^{128 \times 128} \times 12$
Softmax	shift + exp ($\kappa = 2$)	$\mathbf{xw}$	$12 \rightarrow 12$	$16 \rightarrow 12$	$\mathbb{R}^{128 \times 128} \times 12$
	Softmax sum + reciprocal	$\mathbf{xw}$	$12 \rightarrow 12 + 12$ slices	multi-refresh	$\mathbb{R}^{128 \times 128} \times 12$
	normalize	$\mathbf{xw}$	$12 \rightarrow 12$	$12 \rightarrow 6$	$\mathbb{R}^{128 \times 128} \times 12$
Attention out	PV (CCMM, Y -encode fused)	$\mathbf{xw} \rightarrow \mathbf{xyw}$	$24 \rightarrow 12$	$5 \rightarrow 4$	$(\mathbb{R}^{128 \times 128} \times \mathbb{R}^{128 \times 64}) \times 12$
	output projection (PCMM, Y -decode fused)	$\mathbf{xyw} \rightarrow \mathbf{xw}$	$12 \rightarrow 3$ acc.	$4 \rightarrow 3$	$\mathbb{R}^{128 \times 768} \times \mathbb{R}^{768 \times 768}$
	LayerNorm statistics + inv. sqrt	$\mathbf{xw}$	$3 \rightarrow 3 + 1$ slice	$3 \rightarrow 1$; refresh	$\mathbb{R}^{128 \times 768}$
	LayerNorm normalize	$\mathbf{xw}$	$3 \rightarrow 3$	$3 \rightarrow 1$	$\mathbb{R}^{128 \times 768}$
Bootstrap	full-ciphertext refresh	$\mathbf{xw}$	$3 \rightarrow 3$	$1 \rightarrow 16$	$\mathbb{R}^{128 \times 768}$
FFN	FC1 (PCMM)	$\mathbf{xw}$	$3 \rightarrow 12$ str.	$13 \rightarrow 12$	$\mathbb{R}^{128 \times 768} \times \mathbb{R}^{768 \times 3072}$
	GELU ($\kappa = 4$)	$\mathbf{xw}$	$12 \rightarrow 12$ str.	$12 \rightarrow 3$	$\mathbb{R}^{128 \times 3072}$
	FC2 (PCMM)	$\mathbf{xw}$	$12 \rightarrow 3$ acc.	$3 \rightarrow 2$	$\mathbb{R}^{128 \times 3072} \times \mathbb{R}^{3072 \times 768}$
	LayerNorm statistics + inv. sqrt	$\mathbf{xw}$	$3 \rightarrow 3 + 1$ slice	$3 \rightarrow 1$; refresh	$\mathbb{R}^{128 \times 768}$
	LayerNorm normalize	$\mathbf{xw}$	$3 \rightarrow 3$	$3 \rightarrow 1$	$\mathbb{R}^{128 \times 768}$
Bootstrap	full-ciphertext refresh	$\mathbf{xw}$	$3 \rightarrow 3$	$1 \rightarrow 16$	$\mathbb{R}^{128 \times 768}$

degradation in system performance or privacy guarantees. The reported 7.04s amortized latency assumes a batch of 256 records encrypted under one client key. Only the public, offline calibration of approximation intervals is task-specific.

4.4 Memory Footprint Analysis

Table 6 accounts for where the memory goes: it models the theoretical peak of end-to-end BERT-base inference as the levers of Section 3 are enabled cumulatively, at the parameters of this section. A naive GL port is dominated by its pre-encoded plaintexts: BERT-base’s $\approx 5,200$ weight tiles, expanded to full ring elements at their consumption levels, alone occupy about 2 TB. The expansion-free pipeline (Section 3.5) collapses them to their ≈ 0.35 GB cleartext size, leaving the whole-payload bootstrap working set and the matrix-native BSGS scratch as the next walls; the Y -decoded representation (Section 3.4) removes the latter, and slice-tiled execution with shared keyswitch scratch (Section 3.2) brings the bootstrap to its measured 17.4 GB peak. The complete pipeline, including all twelve layers, bootstrapping, and attention, runs at a sampled peak of 29.8 GB, within a single 32 GB consumer GPU with no host streaming.

5 Related Work

We organize related work into three lines: GPU systems for CKKS, encrypted transformer inference on GPUs, and private transformer inference through MPC and hybrid HE-MPC solutions.

GPU-accelerated CKKS. The first generation of GPU CKKS libraries ported the CPU’s 64-bit RNS arithmetic onto CUDA cores and demonstrated large gains over CPU baselines [1, 28, 51]; FHE ASICs mark the far end of this hardware spectrum [43, 44]. A second line repurposed otherwise idle tensor cores [15, 16, 27], culminating in HyperDrive [14], which co-optimizes NTT data movement through the memory hierarchy. A third line instead rebuilt the RNS arithmetic itself around the GPU’s native 32-bit word; we compare in detail against its three systems, which are the closest to ours.

Cheddar [9] made CKKS execution fully 32-bit through a 25–30 bit prime system with rational rescaling, signed Montgomery kernels, and extensive kernel fusion, and holds the fastest published CNN-scale results. Its limits for transformer workloads are structural (Section 2.1): matrix products cost rotation ladders that

Table 5. Per-operation execution time (seconds) of encrypted BERT-base inference. Baselines run on a single A100: THOR [37], Powerformer [39] (retrained model; BPmax and Batch LN on the Softmax and LayerNorm rows), MOAI [53], and Tricycle [33]. Terrazzo is reported on both GPUs for completeness. Terrazzo times are amortized per input over the 256-way packed batch.

Operation	THOR	Powerf.	MOAI	Tricycle	Terrazzo (A100)	Terrazzo (5090)
Attn. layer	49.77	57.65	16.54	21.00	0.57	0.19
Attn. score	16.25	28.76	14.07	2.99	0.15	0.03
Softmax	15.53	0.75	2.19	3.96	1.48	0.34
Attn. heads	13.08	18.95	0.29	1.68	7.04	1.30
MHA	27.43	22.54	0.48	4.96	0.14	0.05
LayerNorm1	7.13	0.37	0.15	4.93	0.03	0.03
FC1	49.80	59.21	16.18	17.52	0.30	0.23
GELU	29.42	8.31	3.30	7.26	15.30	3.47
FC2	49.19	43.77	2.88	12.33	0.19	0.16
LayerNorm2	4.10	0.30	0.15	4.94	0.04	0.03
Pooler + cls.	2.70	0.20	0.70	—	0.09	0.03
Bootstraps	337.86	103.72	227.84	18.14	4.65	1.19
Total	602.26	344.52	283.95	100.55	29.98	7.04

Table 6. Theoretical peak device memory of end-to-end BERT-base inference as memory levers are enabled cumulatively, at the parameters of Section 4. Modeled rows are analytic; the final row is the measured sampled peak.

Configuration	Peak memory
Naive port (pre-expanded plaintexts, xyw only)	≈ 2.1 TB
+ expansion-free plaintexts (§3.5)	≈ 105 GB
+ Y-decoded bootstrap and nonlinear layers (§3.4)	≈ 75 GB
+ slice tiling and shared scratch (§3.2)	≈ 32 GB
Full pipeline (measured)	29.8 GB

fusion cannot remove, and its scheduling unit is the whole ciphertext. Terrazzo keeps its word-size discipline through grafting (Section 3.3), executes matrix products as rotation-free GEMMs, and schedules below the ciphertext (Section 3.2); Table 2 compares bootstrapping per RLWE slice.

Theodosian [10] contributed the memory-wall analysis this paper builds on (Section 2.4); its authors conclude that further progress must shift the memory wall itself through cryptographic enhancements. Terrazzo is an answer to that call. Where Theodosian orchestrates a fixed set of bytes, our Y-decoded representation, expansion-free plaintexts, and seeded keys shrink the bytes themselves; and κ -tiling is L2-aware batching pushed *inside* a single homomorphic operation.

Zephyr [45] is the system closest to our modulus-management contribution. It builds a 30-graft CKKS system on grafting [6]: the RNS basis consists entirely of 30-bit primes, and the scale is managed through an auxiliary graft modulus, at the cost its authors identify: the graft modulus participates in every computation. This is precisely the cost Terrazzo confines: grafting is restricted to the application levels, where an off-unit divisor requires it, and bootstrapping, the deepest circuit in the system, runs sprout-free. Zephyr reaches BERT-base in 107.9s on an A100 and fits a 24 GB RTX 4090, but reports no standalone bootstrap latency, so it appears in Table 3 but not in Table 2.

A complementary line optimizes above individual kernels at compile time [2, 54]; its most ambitious system is Cerium [25], a DSL, compiler, and runtime for encrypted large-model inference: IR-level kernel fusion, host-device memory orchestration, and multi-GPU parallelization over a 28-bit RNS basis reach BERT-base in 66s on one A100. These systems attack a different layer than Terrazzo: they orchestrate the bytes that a fixed scheme representation dictates. Terrazzo instead uses a sub- 2^{31} prime basis with grafted

35-bit application scales and shrinks the representation itself. It evaluates an unmodified BERT-base in 29.98s on one A100 without host streaming.

Encrypted transformer inference on GPUs. The systems we compare against in Tables 3 and 5 each attack CKKS’s packing problem with a bespoke encoding: diagonal-major packing in THOR [37], module-optimized packing with interleaved batching in MOAI [53], tricyclic encodings with a statistical-max Softmax in Tri-cycle [33], or by retraining the model into FHE-friendly form, as in Powerformer’s distilled BPmax and Batch LN [39]. Terrazzo instead uses native matrix multiplication, fuses representation switches into adjacent weight operators, evaluates nonlinear layers over small slice tiles, and refreshes 128 RLWE slices in one bootstrap. NEXUS, Euston, and other adjacent systems use different models, hardware scales, or reporting settings [12, 17, 19, 21, 42, 48, 52], so Table 3 compares per-input, end-to-end BERT-base latency on one A100, with Terrazzo amortized over its native 256-input batch.

Private transformer inference through MPC and hybrid HE-MPC solutions. MPC/FSS designs have advanced rapidly [5, 13, 20, 29, 35, 50], and a parallel hybrid line evaluates linear layers under HE and nonlinear ones under MPC [24, 32, 34, 38, 49, 55]. Among these, the systems reporting end-to-end GPU timings for BERT-base include SHAFT, at 28.6s in a LAN setting on two A40s, and BLB, at 2.5 min on an A100 in a LAN setting, growing to 13.2 min in its slowest WAN setting; SIGMA’s 1.72s is online time only, after dealer preprocessing and a 16.84 GB per-inference key transfer. These systems achieve competitive latency but require interaction, network-dependent communication, and sometimes trusted preprocessing [20]. Terrazzo instead provides single-server, non-interactive inference with one upload and one download, although its 7.04-second RTX 5090 latency is amortized over a native 256-input batch.

6 Conclusion

The GL scheme removes the central mismatch between CKKS and transformer workloads: matrix multiplication becomes a native homomorphic operation, but its ciphertexts, plaintexts, and evaluation keys are large enough that GPU deployment has appeared impractical. Terrazzo shows that the obstacle is not fundamental. By bootstrapping into a compact Y-decoded representation, tiling every kernel over independently schedulable ciphertext slices, confining grafting to application levels around a sprout-free bootstrap, and expanding every plaintext from its cleartext footprint inside the consuming kernel, Terrazzo achieves a 7.04 s amortized time per input for unmodified BERT-base on a consumer 32 GB RTX 5090. Its A100 time is 29.98 s amortized, a $2.20\text{--}20.09\times$ speedup over prior single-A100 systems. Per-slice bootstrap speedup is $1.12\text{--}2.03\times$ on the RTX 5090 and $1.08\text{--}1.36\times$ on the A100. The broader lesson is that the FHE memory wall moves when the representation moves: co-designing what the scheme stores with what the GPU schedules yields gains that kernel tuning alone cannot. As future work, we plan to extend Terrazzo to decoder-only models and to multi-GPU scaling.

Acknowledgment

The authors would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

1. Carlos Agulló-Domingo, Óscar Vera-López, Seyda Guzelhan, Lohit Daksha, Aymane El Jerari, Kaustubh Shivdikar, Rashmi Agrawal, David Kaeli, Ajay Joshi, and José L Abellán. Fideslib: A fully-fledged open-source fhe library for efficient ckks on gpus. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2025.
2. Song Bian, Yintai Sun, Zian Zhao, Haowen Pan, Mingzhe Zhang, and Zhenyu Guan. Libra: Pattern-scheduling co-optimization for cross-scheme fhe code generation over gpgpu. In *USENIX Security Symposium*, 2026.

3. Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. Security guidelines for implementing homomorphic encryption. *IACR Communications in Cryptology*, 1(4), 2025.
4. Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–54. Springer, 2019.
5. Ke Cheng, Yuheng Xia, Anxiao Song, Jiaxuan Fu, Wenjie Qu, and Yulong Shen. Mosformer: Maliciously secure three-party inference framework for large transformers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.
6. Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeon Noh. Grafting: decoupled scale factors and modulus in rns-ckks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1098–1112, 2025.
7. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.
8. Wonhee Cho, Guillaume Hanrot, Taeseong Kim, Minje Park, and Damien Stehlé. Fast and accurate homomorphic softmax evaluation. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.
9. Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library designed for gpu architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 35–49, 2026.
10. Wonseok Choi, Hyunah Yu, Jongmin Kim, Hyesung Ji, Jaiyoung Park, and Jung Ho Ahn. Theodosian: A deep dive into memory-hierarchy-centric fhe acceleration. In *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 299–310. IEEE, 2026.
11. Eric Crockett, Craig Gentry, Hyojun Kim, Yeongmin Lee, and Yongwoo Lee. Efficient bootstrapping in fully homomorphic encryption for matrix arithmetic. In *Annual International Cryptology Conference*, pages 492–524. Springer, 2026.
12. 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 *International Conference on Machine Learning*, 2025.
13. 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.
14. Guang Fan, Yi Chen, Lei Chen, Liang Kong, Chao Niu, Dian Jiao, Yilan Zhu, Geng Yang, Shengyu Fan, Xianglong Deng, Fangyu Zheng, Jian Weng, Yisong Chang, Meng Li, Shoumeng Yan, and Mingzhe Zhang. Hyperdrive: Hierarchical exploitation of memory efficiency for gpu-based fhe acceleration. In *International Symposium on Computer Architecture*, 2026.
15. 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 *IEEE International Symposium on High-Performance Computer Architecture*, 2025.
16. Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *IEEE International Symposium on High-Performance Computer Architecture*, 2023.
17. Xinwen Gao, Shaojing Fu, Lin Liu, Zhuotao Liu, Yuchuan Luo, and Yongjun Wang. Euston: Efficient and user-friendly secure transformer inference with non-interactivity. *Cryptology ePrint Archive*, 2026.
18. Craig Gentry and Yongwoo Lee. Fully homomorphic encryption for matrix arithmetic. In *Annual International Cryptology Conference*, pages 363–395. Springer, 2026.
19. Zhaoting Gong, Ran Ran, Fan Yao, and Wujie Wen. Scaling long-sequence homomorphic encrypted transformer inference via hybrid parallelism on multi-gpu systems. In *Proceedings of the 40th ACM International Conference on Supercomputing*, 2026.
20. Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. *Proceedings on Privacy Enhancing Technologies*, 2024.
21. Seyda Nur Guzelhan, Lohit Daksha, Carlos Agulló Domingo, Gilbert Jonatan, John Kim, José L Abellán, David Kaeli, and Ajay Joshi. Ellmo: Packing- and depth-aware encrypted transformer inference, 2026. Preprint.
22. Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual International Cryptology Conference*, pages 554–571. Springer, 2014.

23. Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 364–390. Springer, 2020.
24. Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In *Advances in Neural Information Processing Systems*, 2022.
25. 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.
26. Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1209–1222, 2018.
27. 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 *International Symposium on Computer Architecture*, 2025.
28. 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.
29. Andes Y L Kei and Sherman S M Chow. Shaft: Secure, handy, accurate and fast transformer inference. In *Network and Distributed System Security Symposium*, 2025.
30. Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 648–677. Springer, 2021.
31. Baiyu Li, Daniele Micciancio, Mark Schultz, and Jessica Sorrell. Securing approximate homomorphic encryption using differential privacy. In *Annual International Cryptology Conference*, pages 560–589. Springer, 2022.
32. Zhengyi Li, Kang Yang, Jin Tan, Wen-jie Lu, Haoqi Wu, Xiao Wang, Yu Yu, Derun Zhao, Yancheng Zheng, Minyi Guo, and Jingwen Leng. Nimbus: Secure and efficient two-party inference for transformers. In *Advances in Neural Information Processing Systems*, 2024.
33. Lawrence Lim, Vikas Kalagi, Julia Novick, Jiaming Liu, Divyakant Agrawal, and Amr El Abbadi. Tricycle: Private transformer inference with tricyclic encodings. *Cryptology ePrint Archive*, 2025.
34. 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 *Network and Distributed System Security Symposium*, 2025.
35. Yunlv Lv, Rui Zhang, Zhiyuan Zhang, Ziyi Wan, Lanxue Zhang, Minhui Xue, Jiangtao Li, and Yanan Cao. Smash: Scalable maliciously secure hybrid multi-party computation framework for privacy-preserving large language models. In *USENIX Security Symposium*, 2026.
36. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 1–23. Springer, 2010.
37. 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*, 2025.
38. Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *IEEE Symposium on Security and Privacy*, 2024.
39. Dongjin Park, Eunsang Lee, and Joon-Woo Lee. Powerformer: Efficient and high-accuracy privacy-preserving language model with homomorphic encryption. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11090–11111, 2025.
40. Jaiyoung Park, Sejin Park, Jai Hyun Park, Jung Ho Ahn, Jung Hee Cheon, Guillaume Hanrot, Jung Woo Kim, Minje Park, and Damien Stehlé. Scaling up privacy-preserving ml: A ckks implementation of llama-2-7b. *arXiv preprint arXiv:2601.18511*, 2026.
41. Michael S Paterson and Larry J Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.
42. Donghwan Rho, Taeseong Kim, Minje Park, Jung Woo Kim, Hyunsik Chae, Ernest K Ryu, and Jung Hee Cheon. Encryption-friendly llm architecture. In *International Conference on Learning Representations*, 2025.
43. Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *IEEE/ACM International Symposium on Microarchitecture*, 2021.
44. Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data. In *International Symposium on Computer Architecture*, 2022.

45. Sieun Seo and Chohong Min. Zephyr: Gpu-efficient homomorphic encryption for privacy-preserving transformer inference. *Cryptology ePrint Archive*, 2026.
46. Guiming Shi, Yuchen Wei, Shengyu Fan, Xianglong Deng, Liang Kong, Xianbin Li, Jingwei Cai, Shuwen Deng, Mingzhe Zhang, and Kaisheng Ma. Wpc: Weight plaintext compression for cnn inference based on rns-ckks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 4094–4108, 2025.
47. Rostin Shokri and Nektarios Georgios Tsoutsos. Efficient bootstrapping of matrices in fhe. *Cryptology ePrint Archive*, 2026.
48. Jianhang Xie, Sicheng Tan, Vishnu Naresh Boddeti, and Zhichao Lu. Atlas: Automated approximation of transformers for efficient homomorphic inference in one hour, 2026. Preprint.
49. Tianshi Xu, Wen-jie Lu, Jiangrui Yu, Yi Chen, Chenqi Lin, Runsheng Wang, and Meng Li. Breaking the layer barrier: Remodeling private transformer inference with hybrid ckks and mpc. In *USENIX Security Symposium*, 2025.
50. Guang Yan, Yuhui Zhang, Zimu Guo, Lutan Zhao, Xiaojun Chen, Chen Wang, Wenhao Wang, Dan Meng, and Rui Hou. Comet: Accelerating private inference for large language model by predicting activation sparsity. In *IEEE Symposium on Security and Privacy*, 2025.
51. 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.
52. 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*, 2025.
53. 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. *Cryptology ePrint Archive*, 2025.
54. Jianyu Zhao, Xueyu Wu, Guang Fan, Mingzhe Zhang, Shoumeng Yan, Lei Ju, and Zhuoran Ji. A framework for developing and optimizing fully homomorphic encryption programs on gpus. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2026.
55. Yufan Zhu, Chao Jin, Khin Mi Mi Aung, and Xiaokui Xiao. Encformer: Secure and efficient transformer inference over encrypted data. *arXiv preprint arXiv:2604.09975*, 2026.

A Notations and Conventions

Vectors are written in bold lowercase ($\mathbf{v}$) and matrices in bold uppercase ($\mathbf{M}$). All logarithms are base 2. For a modulus q , $[\cdot]_q$ denotes the centered representative in $(-q/2, q/2]$. We write $\mathcal{O}(\cdot)$ for asymptotic bounds. We largely borrow the notation of the GL bootstrapping paper [47] for scheme material and of Cheddar [9] for CKKS and GPU material. Table 7 summarizes the symbols used throughout the paper.

Table 7. Notation used throughout the paper.

Symbol	Description
$\mathcal{R}'$	GL plaintext ring: $\mathbb{Z}[i][X, Y, W]/\langle X^n - i, Y^n - i, \Phi_p(W) \rangle$
$\mathcal{R}$	Y -free base ring $\mathbb{Z}[i][X, W]/\langle X^n - i, \Phi_p(W) \rangle$
$\mathcal{R}_{\text{aux}}$	auxiliary ring adjoining a second W -type variable V
n	X/Y -axis dimension
p, ϕ	W -axis prime and $\phi = \varphi(p) = p - 1$
$s(X, W)$	secret key, drawn from $\mathcal{R}$ (“ Y -free”)
$\text{ct} = (b, a)$	GL ciphertext pair over $\mathcal{R}'_{Q_\ell}$
q_i	RNS primes, $q_i \equiv 1 \pmod{4np}$
α	exponent of the power-of-two graft sprout 2^α
Q_ℓ	ciphertext modulus $2^\alpha \prod_{i \leq \ell} q_i$
P	auxiliary (special) modulus for keyswitching
dnum	number of keyswitching digits
evk	evaluation key; small over $\mathcal{R}_{PQ_L}$, big over $\mathcal{R}'_{PQ_L}$
L, ℓ	maximum and current multiplicative level
Δ	message scaling factor
$\mathbf{M}^{(w)} \in \mathbb{C}^{n \times n}$	message matrix in the w -th W slot
$\otimes$	native GL matrix product (trace contraction, §2.1)
N	CKKS ring degree (baselines; typically 2^{16})
κ	tile size: Y slices processed together, $1 \leq \kappa \leq n$
d	degree of an approximating polynomial
B	BSGS baby-step count