ELLMo: Encrypted Transformers with Efficient Linear Algebra and Automated Approximations

Anonymous authors
Paper under double-blind review

Abstract

Cloud-based large language model inference exposes sensitive inputs in plaintext. Fully Homomorphic Encryption removes this exposure by computing on encrypted inputs, but its cost keeps encrypted inference impractical for billion-scale models. Two overheads dominate: i) rotations for indexing elements in matrix multiplication in the linear layers, and ii) frequent bootstrapping to support nonlinearities through polynomial approximations. We present ELLMo, a framework for encrypted transformer inference that optimizes both linear and nonlinear layers. First, ELLMo proposes $\mathrm{ShallowPCMM}$, a novel plaintext-ciphertext matrix multiplication algorithm that reduces rotations and weight memory. Second, ELLMo automates parameter selection for nonlinear layers: it emulates encrypted inference to select the lowest precision the model can tolerate, measures per-layer input ranges, then chooses nonlinear approximations that minimize the number of bootstrapped ciphertexts. On a single A100, ELLMo classifies one encrypted 128-token input in under 58 s with BERT-Base and 625 s with Llama-3-8B. With ELLMo, Llama-3-8B decodes one token in 80 s to 101 s at context lengths of 256 to 512, up to 3.1× faster than prior work. Encrypted accuracy and teacher-forced perplexity stay within 0.35% of plaintext on every model. On BERT-Base, ELLMo is 1.1× to 12× faster than prior unamortized systems, and needs 3.2× to 5.2× less weight memory. Source code will be released upon publication.

1 Introduction

Enterprises run Large Language Models (LLMs) in the cloud, where their queries, often sensitive, are exposed in plaintext to the provider. Fully Homomorphic Encryption (FHE) eliminates this exposure by enabling inference directly on encrypted inputs (Gentry, 2009). The cloud computes on the encrypted data without decrypting it and returns an encrypted result that only the data owner can decrypt. Among the FHE schemes, Cheon-Kim-Kim-Song (CKKS) (Cheon et al., 2017) is the primary choice for privacy-preserving inference, because it supports approximate single-instruction-multiple-data arithmetic on real and complex numbers.

Unfortunately, encrypted LLM inference costs up to four orders of magnitude more compute and memory than plaintext inference, which hinders practical deployment (Al Badawi et al., 2026). Two costs dominate encrypted transformer inference under CKKS. The first is matrix multiplication, whose many rotations each require an expensive $\mathrm{Key\text{-}Switch}$ operation. Its multiplicative depth also sets the level at which the weights are encoded, and the memory of an encoded weight grows with that level. The second is bootstrapping, which refreshes a ciphertext after a fixed number of sequential multiplications (Cheon et al., 2018). Nonlinearities such as Softmax are approximated using high-degree polynomials in CKKS, requiring frequent bootstraps (Cho et al., 2024). Together, matrix multiplications and bootstraps take up to 93% of the inference time (Moon et al., 2025).

We propose ELLMo, an automated framework for encrypted transformer inference that addresses both costs. We target a setting where the user data is encrypted under FHE, and the server computes on it with a plaintext model (Gilad-Bachrach et al., 2016), such as an open-weight LLM. ELLMo has two components: a new matrix multiplication algorithm and an offline calibrator for the nonlinear layers.

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

Efficient encrypted matrix multiplication. In this setting, a linear layer multiplies encrypted activations with plaintext weights, which is a plaintext-ciphertext matrix multiplication (PCMM). For an $M \times K$ activation matrix and a $K \times N$ weight, a PCMM needs $O(\sqrt{d})$ rotations when $M = K = N = d$ (Halevi & Shoup, 2018), and each rotation invokes a $\mathrm{Key\text{-}Switch}$. Prior algorithms reduce the rotation count with extra multiplicative depth (Moon et al., 2025). Extra depth encodes the weights at higher levels, where each plaintext has more limbs and takes more memory, and it uses up levels sooner, so the model needs more bootstraps. ELLMo introduces $\mathrm{ShallowPCMM}$ (SPCMM, Section 3), which reduces rotations and consumes at most two multiplicative levels. ELLMo runs every PCMM at the lowest levels, where each weight plaintext has few limbs. SPCMM moves part of the computation on the weights offline, and sparse encoding (Jayashankar et al., 2025) keeps these precomputed weights at the same amount of memory as the original encoded weights.

Automated nonlinear calibration. CKKS evaluates each nonlinearity as a polynomial with a fit interval and a degree, and reciprocals and inverse square roots also need a number of Newton-Raphson (NR) iterations. Tuning these by hand is not practical, because activation ranges differ by orders of magnitude across the layers of a model and also change across models. ATLAS (Xie et al., 2026) automates this choice, but it minimizes the depth of the approximations and places bootstraps afterward. Depth alone does not set the cost, because a bootstrap refreshes one ciphertext and the number of ciphertexts to refresh varies across the model, so the cost to minimize is the number of bootstrapped ciphertexts. ATLAS also sets each fit interval to the range measured on a small data subset, which leaves little room for inputs outside that range. The ELLMo calibrator instead picks, for each nonlinearity, the approximation with the fewest bootstrapped ciphertexts that keeps the encrypted logits within a tolerance of the plaintext logits. It also widens each fit interval to leave room for inputs outside the calibration data, and it guarantees that the NR iterations converge on the whole interval. Calibration runs once per model, offline, on a public dataset, takes under one hour, and never sees encrypted user inputs.

Results. We implement ELLMo using the open-source GPU library FIDESlib (Agulló-Domingo et al., 2025) and run end-to-end encrypted inference on BERT-Base, GPT-2, Llama-3.2-1B, and Llama-3-8B on a single NVIDIA A100 and report unamortized latency on a single 128-token input. On BERT-Base, ELLMo is 1.1× faster than CERIUM (Jayashankar et al., 2025), which relies on compiler optimizations, 2.0× faster than Tricycle, and 5.7× faster than MOAI (Zhang et al., 2026), which amortizes latency over a batch. On Llama-3-8B, ELLMo is 1.1× faster than CERIUM on prefill, comparable to Cachemir (Yu et al., 2026b), and 3.1× faster than ROSETTA (Yu et al., 2026a) on decoding. ELLMo needs 3.2× to 5.2× less weight memory than CERIUM. Across all models, encrypted accuracy and teacher-forced perplexity stay within 0.35% of plaintext.

Related Work. Several prior works implement encrypted transformer inference by modifying the model (Rho et al., 2025; Park et al., 2025; Wang et al., 2026) or by batching several inputs (Gao et al., 2026; Zhang et al., 2026; 2025). A third line of work, including ELLMo, keeps the original model and runs one input at a time. For linear layers, THOR uses diagonal packing (Moon et al., 2025), Powerformer uses column packing (Park et al., 2025), and Tricycle uses tricyclic encodings (Lim et al., 2025). For nonlinear layers, CERIUM uses compiler optimizations for bootstrapping (Jayashankar et al., 2025), ATLAS searches approximations automatically (Xie et al., 2026), and Cachemir adds an encrypted KV cache for generation (Yu et al., 2026b). On Llama-3-8B, prior work reports only latency (Jayashankar et al., 2025; Zhang et al., 2026), evaluates a BERT-style block of the same size (Zhang et al., 2025; Gao et al., 2026; Wang et al., 2026), or measures accuracy or perplexity in plaintext, either with approximated nonlinearities (Yu et al., 2026b;a) or with modeled CKKS noise (Park et al., 2026). In this paper, we run the unmodified Llama-3-8B end to end under encryption and report its encrypted accuracy and teacher-forced perplexity along with its latency, which to our knowledge no prior work does.

2 PRELIMINARIES

Fully Homomorphic Encryption. In this paragraph, we provide a short FHE refresher for readers familiar with FHE and provide a detailed description in Appendix A.1. FHE schemes are classified by the computation they support: TFHE and FHEW for Boolean logic, BGV and BFV for exact integer arithmetic, and CKKS for approximate arithmetic over complex numbers (Marcolla et al.,

Table 1: Key notation used throughout the paper.

FHE		Transformers		Matrix Multiplication	
N	Ring degree.	T	Sequence length.	$\times$	Matrix product.
n	Number of slots.	D	Hidden dimension.	$\odot$	Hadamard product.
Q	Ciphertext modulus.	$d \times d$	Square packed tile size.	τ	Lower diagonal transform.
ℓ / L	Current / maximum level.	d_h	Attention head dimension.	ϕ^k	Rotate by k columns.
L_{eff}	Post-bootstrap depth.	H	Number of attention heads.	ζ^k	Select k^{th} row.

2022). We target the CKKS scheme, and Table 1 lists the notation used in the paper. CKKS operates on a polynomial ring of degree N and modulus Q, where the pair (N, Q) must satisfy the security requirements (Bossuat et al., 2024). In practice, CKKS uses a Residue Number System (RNS) variant, where the modulus is a product of primes $Q = q_0 \cdot q_1 \cdots q_L$, and a polynomial at level ℓ is stored as $\ell + 1$ residue polynomials called limbs (Cheon et al., 2019). A ciphertext or plaintext at level ℓ has $\ell + 1$ limbs of N coefficients each, so its memory scales with $(\ell + 1) \cdot N$. For example, with $N = 2^{16}$ and 64-bit words, a plaintext takes 1.5 MB at $\ell = 2$ and 6 MB at $\ell = 11$.

The server evaluates Add, Mult, and Rotate on ciphertexts. A product of two ciphertexts and a Rotate leave the result encrypted under a different key, and a $\mathrm{Key\text{-}Switch}$ routine brings it back to the original secret key using a public evaluation key. This routine costs an order of magnitude more than the other basic operations (Agrawal & Joshi, 2023). Its cost is dominated by two modulus conversions, a ModUp at the start and a $\mathrm{ModDown}$ at the end, and double hoisting amortizes both (Bossuat et al., 2021). Rotations of the same ciphertext share one ModUp, and a sum of rotated ciphertexts needs only one $\mathrm{ModDown}$. Each Mult squares the scale of the encoded values, and a $\mathrm{Rescale}$ restores it by dropping one limb, so a ciphertext supports a limited number of sequential multiplications. Finally, the $\mathrm{Bootstrap}$ operation restores the levels and is itself a long chain of Mult and Rotate calls, which makes it the most expensive routine in FHE (Kim et al., 2024).

Encrypted Inference of Transformers. We evaluate ELLMo on BERT, GPT-2, and Llama, which we summarize below. A transformer layer consists of Multi-Head Attention (MHA) and a Feed-Forward Network (FFN), which act on a token matrix $X \in \mathbb{R}^{T \times D}$ (Vaswani et al., 2017). MHA computes $Q = XW_Q$, $K = XW_K$, and $V = XW_V$, splits them into H heads $\in \mathbb{R}^{T \times d_h}$, and computes $\mathrm{Softmax}(Q_h K_h^\top / \sqrt{d_h})\, V_h$ for each head h. The heads are concatenated and projected by W_O. The FFN expands each token to a larger width, applies a nonlinear activation, and projects back to D. BERT and GPT-2 normalize with LayerNorm, and Llama with RMSNorm (Devlin et al., 2019; Radford et al., 2019). Appendix A.2.1 provides a more elaborate description of transformers.

In encrypted transformers, the linear projections are plaintext-ciphertext matrix multiplications (PCMMs), and the attention score products are ciphertext-ciphertext matrix multiplications (CCMMs). The Jiang-Kim-Lauter-Song (JKLS) method uses linear transforms to reduce the rotations in both (Jiang et al., 2018). For example, transform ϕ operates on a matrix X as $\phi^k(X)_{i,j} = X_{i,\, j+k}$ to shift the columns by k. A $d \times d$ tile is packed row-major into the slots. The transforms can be implemented with the Baby-Step-Giant-Step (BSGS) algorithm, which decomposes rotations of $r \in \{0, \ldots, d-1\}$ into baby steps b and giant steps g, where $b \approx g \approx \sqrt{d}$ (Halevi & Shoup, 2018). Under CKKS, Softmax needs an exponential and a reciprocal, the normalizations need an inverse square root, and all nonlinearities are evaluated with high-degree polynomials (Zhang et al., 2025). Readers new to encrypted transformer inference can find a detailed version in Appendix A.2.2.

3 Efficient Encrypted Linear Layers

The cost of an encrypted matrix multiplication comes from three sources: the number of rotations, whether those rotations can be hoisted, and the multiplicative depth, which sets the level and memory of the weights (Section 2). We propose $\mathrm{ShallowPCMM}$ (SPCMM), which reduces rotations and depth together while preserving hoisting (Section 3.1). We then combine SPCMM with a sparse encoding, which stores its precomputed weights at no extra memory cost. For the ciphertext-ciphertext products of attention, we reduce rotations and reuse transformed operands across tiles (Section 3.2). We then fuse complex packing into the existing matrix multiplication algorithms to reduce the number of ciphertexts at no extra depth (Section 3.3).

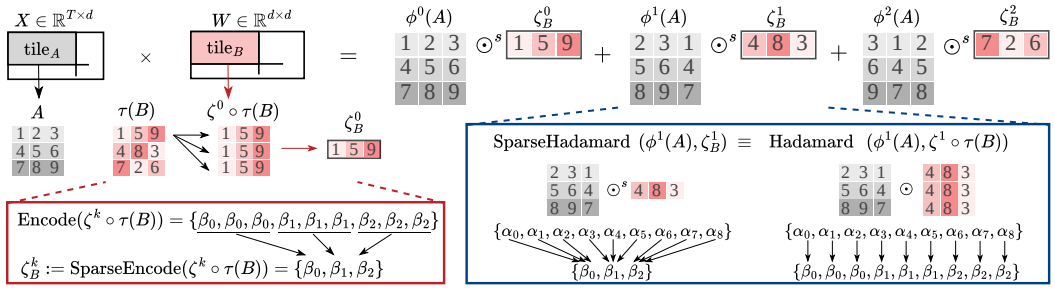


Figure 1: SPCMM with sparse encoding for $d = 3$. An encrypted tile A of the activation X is multiplied by a plaintext tile B of the weight W as $\sum_k \phi^k(A) \odot^s \zeta_B^k$, where ζ_B^k is the sparse encoding of $\zeta^k \circ \tau(B)$. Every row of $\zeta^k \circ \tau(B)$ is row k of $\tau(B)$, so each NTT value β_j fills one run of consecutive positions (red box). SparseEncode keeps one copy per run, which cuts the plaintext storage from N to N/d values per limb. SparseHadamard ($\odot^s$) multiplies each run of the ciphertext by one stored value and gives the same result as the dense Hadamard product (blue box). The figure is schematic: it draws each ζ_B^k as row k of $\tau(B)$ instead of its NTT values, and in CKKS d is a power of two.

3.1 Shallow PCMM

SPCMM reduces the rotation cost by applying only column shifts to the ciphertext which are hoistable and a row-repeating transform to the plaintext. For simplicity, we present matrix tiles as squares of dimension $d \times d$. Appendix A.3.1 gives the proofs of the SPCMM proposition and lemmas and Appendix A.3.2 generalizes the $d \times d$ case to rectangular tiles.

Proposition 3.1 (SPCMM)**.** Let $X_{i,j}$ be the entry of a matrix X in its ith row and jth column. Use $\phi^k(X)_{i,j} = X_{i,\,j+k}$ and $\tau(X)_{i,j} = X_{i+j,\,j}$ from JKLS (Jiang et al., 2018), with indices taken modulo d, and define the row-select transform $\zeta^k(X)_{i,j} = X_{k,j}$. Then the product of a ciphertext $A \in \mathbb{R}^{d\times d}$ and a plaintext $B \in \mathbb{R}^{d\times d}$ can be computed as

$$A \times B = \sum_{k=0}^{d-1} \phi^k(A) \odot \big(\zeta^k \circ \tau(B)\big) \tag{1}$$

All ϕ^k act on the same ciphertext A, so BSGS computes them with only $O(\sqrt{d})$ rotations, and its baby steps can be hoisted to share a single ModUp.

Sparse Encoding. The ζ transform of SPCMM has an important property: every row of $\zeta^k \circ \tau(B)$ equals row k of $\tau(B)$, so the plaintext repeats one row d times. Plaintexts whose slots repeat with a power-of-two period can be stored in compressed form in the NTT domain, as in CERIUM (Jayashankar et al., 2025). The plaintexts of SPCMM have this structure by construction, so all d of them fit in the memory of one dense tile, and we formalize the property below. A ciphertext holds the d rows of a tile in its n slots, so each row spans $c = n/d$ slots, and the slot vector of $\zeta^k \circ \tau(B)$ is c-periodic. This structure appears in both domains: in the coefficient domain, only every d-th coefficient is nonzero, and in the evaluation (NTT) domain, each value repeats d times. Either form can be stored in N/d values per limb. We adopt the evaluation domain, consistent with FIDESlib (Agulló-Domingo et al., 2025), in which the d copies occupy contiguous positions.

Lemma 3.1 (SparseEncode (Jayashankar et al., 2025))**.** *Let $z \in \mathbb{C}^n$ be c-periodic, and let $d = n/c$. Then every NTT limb of* Encode(z) *consists of $2c$ runs, and each run is one value repeated d times:*

$$\{\underbrace{b_0, \ldots, b_{c-1}}_{\text{copy } 0}, \ldots, \underbrace{b_0, \ldots, b_{c-1}}_{\text{copy } d-1}\} \xrightarrow{\text{Encode, NTT}} \{\underbrace{\beta_0, \ldots, \beta_0}_{d}, \ldots, \underbrace{\beta_{2c-1}, \ldots, \beta_{2c-1}}_{d}\} \tag{2}$$

Thus, we store only SparseEncode$(z) := \{\beta_0, \ldots, \beta_{2c-1}\}$ *per limb. The red box of Figure 1 shows these runs.*

SPCMM requires d different plaintexts, one per term of Equation 1, and a dense plaintext stores N values per limb. With SparseEncode, each limb is stored with only $2c = N/d$ values, so the

d plaintexts together take $d \cdot \frac{N}{d} = N$ values, the same size as a single dense encoding of B. The transformed weights can thus be precomputed offline at no memory cost over storing B itself.

Finally, we define the operation that multiplies a ciphertext with a sparse plaintext.

Lemma 3.2 ($\mathrm{SparseHadamard}$ $(\odot^s)$)**.** *Let A be a ciphertext, and let $B' = \mathrm{SparseEncode}(B)$ for a plaintext B with c-periodic slots. $A \odot^s B'$ multiplies run r of every NTT limb of A by β_r of the same limb. The result equals the dense product $A \odot B$. The blue box of Figure 1 illustrates* $\mathrm{SparseHadamard}$*.*

We implement SPCMM as in Proposition 3.2.

Proposition 3.2 (SPCMM with Sparse Encoding)**.** For $d \times d$ tiles A (ciphertext) and B (plaintext), with d a power of two that divides n,

$$A \times B = \sum_{k=0}^{d-1} \phi^k(A) \odot^s \zeta_B^k, \quad \text{where } \zeta_B^k = \mathrm{SparseEncode}(\zeta^k \circ \tau(B)). \tag{3}$$

Cost. SPCMM requires two multiplicative levels with lazy rescaling, and requires $2(b-1)+g$ rotations using BSGS decomposition with $b\,g = d$ and $b \approx g \approx \sqrt{d}$. Each ϕ^k wraps around inside a row, which one slot rotation cannot express, so we rotate twice and mask the two halves. The second rotation is shared across all k within a giant step, which is why the count is $2(b-1)+g$ rather than $2(b+g)$. These masks depend only on the column index, so they keep the row repetition and fold into the precomputed plaintexts, which leaves the sparse encoding applicable. Hoisting shares a single ModUp over the b baby steps. All weight tiles in a row of the product multiply the same rotated copies of A, so this ModUp and the baby steps are paid once per input tile, and only the g giant steps are paid per output tile. A layer with n_{in} input tiles and n_{out} output tiles therefore needs $2(b-1)\,n_{\mathrm{in}} + g\,n_{\mathrm{out}}$ Key-Switch calls. A plaintext at level ℓ occupies $\ell+1$ limbs, so the shallow depth of SPCMM lets the weights sit low in the chain, which shrinks their memory footprint.

3.2 Ciphertext-Ciphertext Matrix Multiplication

Attention forms the two products $QK^\top$ and $\mathrm{Softmax}(QK^\top)\,V$ as ciphertext-ciphertext matrix multiplications (CCMM). We use the blueprint in JKLS (Jiang et al., 2018), which runs CCMM in two stages. The first applies one linear transform per operand, $\sigma(A)_{i,j} = A_{i,\,i+j}$ to the ciphertext on the left and $\tau(B)_{i,j} = B_{i+j,\,j}$ to the one on the right, each as a BSGS diagonal transform. The second stage accumulates d Hadamard products $\phi^k(\sigma(A)) \odot \rho^k(\tau(B))$, where ϕ^k shifts columns by k and ρ^k shifts rows by k. Unlike SPCMM, both operands are rotated, and the first stage costs a level of its own, so CCMM is the deeper of the two products. We lower its cost in three ways. First, each attention head is only $d_h \le d/2$ wide, so a head block leaves half of its columns empty. Folding the data half onto the empty half covers all d second-stage products in a single pass over $d/2$, and packing a second head into the imaginary channel halves this again to $d/4$ products per tile. Second, a row of tiles of A meets every column of tiles of B, so we apply the first-stage transform of that row once and reuse it across the row. Third, we apply double hoisting: one ModUp is shared across the baby steps, and the products of a giant step are accumulated before a single $\mathrm{ModDown}$.

Cost. The first stage applies two BSGS transforms with $2(b-1)+g$ rotations each. The head fold and imaginary packing cut the second stage from d to about $d/4 + O(\sqrt{d})$ rotations, and per-row reuse and double hoisting leave one ModUp and one $\mathrm{ModDown}$ per giant step. The transforms consume three levels, and one mask for the head fold and imaginary packing brings the total to four.

3.3 Complex Packing

Prior work (Park et al., 2025) packs two real matrices into one ciphertext as $A + iB$ and uses $[A \mid B] \times [C \mid D]^\top = \mathrm{Re}\left((A+iB) \times (C-iD)^\top\right) = AC^\top + BD^\top$, which computes a product over $2d$ columns with the work of one over d. It also packs the outputs before bootstrapping, but with a separate step that costs an extra level. Our matrix multiplications instead emit two real outputs $[E \mid F]$ directly as $E + iF$, at no extra level, which halves the bootstrapped ciphertexts after every linear layer. The unpack, one conjugation followed by a multiplication by $-i$, runs only when the

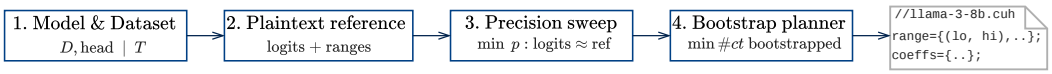


Figure 2: Nonlinear calibration in ELLMo, run offline once per model. (1) The model is exported with hidden dimension D and head type, and the calibration dataset with sequence length T. (2) A plaintext reference run records the logits and the input range of every site. (3) A precision sweep finds the fewest fractional bits p that keep the logits within the tolerance of the reference. (4) The planner picks, for every site, the strategy with the fewest bootstrapped ciphertexts. The output is a per-model header with the range and coefficients of every site.

next layer is nonlinear. Every step that would otherwise cost a level is fused into the precomputed plaintexts: the complex pack and unpack, the column-shift masks, and the scaling factors all ride on the transform plaintexts already present in the BSGS loop.

4 Nonlinear Calibration

CKKS evaluates each nonlinear function either as a direct polynomial or as a polynomial seed refined by Newton-Raphson (NR) iterations, and each choice needs a fit interval, a degree, and, for NR, an iteration count. We call one nonlinear operator in one layer a *site*, for example the Softmax of layer 3 of BERT-Base, and one set of choices for a site a *strategy*, for example a degree-59 Chebyshev polynomial for a GELU site. Every strategy the calibrator considers for a site is a *candidate*. This section describes how ELLMo picks the strategy of every site automatically and offline; Appendix A.4 provides further details.

Why the choice is hard. A higher degree polynomial or more NR iterations lowers the error of an approximation, but it also increases the multiplicative depth. More depth exhausts the levels sooner and forces more bootstraps, so we measure the cost of a strategy by the number of bootstrapped ciphertexts. The goal is the cheapest strategy for every site that keeps the logits within an end-to-end tolerance, by default a mean absolute error of 0.5% relative to the plaintext reference. Manual tuning cannot reach this goal, for three reasons. First, all sites of a model are coupled: an error at one site propagates to every later site, and the levels one site consumes decide where the next bootstrap falls. Second, the search space is large: Llama-3-8B has up to 256 sites with up to 200 candidates each, about 10^{454} combinations (Appendix A.4.1). Third, a strategy does not transfer across models or datasets, because input ranges differ. ELLMo makes the search tractable by removing the coupling in two ways, so each candidate can be scored on its own. First, every PCMM runs at the lowest levels to reduce the memory footprint of weights, so a bootstrap follows every linear layer, simplifying the bootstrapping placement. Second, the calibrator derives a precision target for every site and emulates end-to-end precision.

Calibration pipeline. Figure 2 shows the four steps of the calibrator. Step 1 fixes the set of sites from the model and the calibration dataset. Step 2 runs the plaintext model at its full precision and intercepts each nonlinear operator where the model calls it, so a new architecture needs no new code. It widens each measured range with a tail estimate (Appendix A.4.2), which gives the fit interval directly, so only the discrete choices remain to be searched. Step 3 quantizes the input of every site to p fractional bits, runs the model once per candidate p, and keeps the smallest p that meets the tolerance as the precision target of every site. This is the last step that runs the model, so calibration needs one forward pass per candidate p, plus one for the ranges. Step 4 discards the candidates of each site that miss the target, and picks the one with the fewest bootstrapped ciphertexts (Appendix A.4.3). All four steps run in plaintext, so calibration is cheap: it takes under one hour for every model in Section 5, including Llama-3-8B, on a single A100.

NR-Chebyshev-NR sandwich. The inverse square root of the normalization layers is usually computed from a Chebyshev seed refined by NR iterations, which fails when the input range is wide. A Chebyshev polynomial is accurate only inside its fit interval, and on a wide interval its error is largest near the small end, where the function is steep. NR converges only from a seed close to the answer, so an input past the edge or near the small end makes the iterations diverge and corrupt the output. ELLMo therefore compresses the range before the polynomial: a pre-NR stage of k_1 NR iterations from a constant seed pulls the value the polynomial sees into a narrow window around 1,

Table 2: Matrix multiplication comparison with the state of the art, for a single BERT-Base layer with parameters $T = 128$, $D = 768$, $d_h = 64$, and $H = 12$. #KS is the number of key switches, latencies are in seconds. Numbers in parentheses show the speedup of ELLMo over each baseline.

System	$\times W_{Q/K/V}$		$Q \times K^\top$ & $\times V$		$\times W_O$		$\times W_{FFN1/2}$		Total	
	#KS	s	#KS	s	#KS	s	#KS	s	#KS	s
THOR	192	4.15 (52×)	952	2.44 (6×)	122	2.29 (51×)	536	8.25 (47×)	1802	17.13 (24×)
Powerformer	122	4.80 (60×)	731	3.98 (10×)	75	1.88 (42×)	267	8.58 (49×)	1195	19.24 (27×)
Tricycle	568	1.67 (21×)	572	0.47 (1×)	276	0.41 (9×)	878	2.49 (14×)	2294	5.04 (7×)
ELLMo	198	**0.080**	955	**0.415**	110	**0.045**	408	**0.176**	1671	**0.716**

however wide the input interval is. The calibrator raises k_1 until the seed lies inside the convergence region with a margin for CKKS noise, so the closing NR iterations converge by construction. The pre-NR stage costs $2k_1 + 2$ levels, which Step 4 counts like any other depth (Appendix A.4.4).

5 Experimental Results

ELLMo is implemented using the FIDESlib library (Agulló-Domingo et al., 2025), with the CKKS parameters of $N = 2^{16}$, a 1745-bit ciphertext modulus including the key-switching modulus, $L = 27$, $L_{\text{eff}} = 10$, and $\lambda = 128$ bits, which follow the security guidelines (Bossuat et al., 2024). We evaluate BERT-Base, GPT-2, Llama-3.2-1B, and Llama-3-8B with $T = 128$ input tokens and 128×256 tiles, which fill every slot of a ciphertext. BERT-Base runs four GLUE tasks (Wang et al., 2019), SST-2 (Socher et al., 2013), MRPC (Dolan & Brockett, 2005), RTE (Giampiccolo et al., 2007), and QNLI, the Llama models run SST-2 and MRPC, and GPT-2 and the Llama models run generation on WikiText (Merity et al., 2017). Appendix A.5 lists the model checkpoints and dataset sizes evaluated under encryption. All measurements run on a single NVIDIA A100 80GB. The tile products and attention heads of ELLMo are independent, so they can be distributed across GPUs, as in CERIUM (Jayashankar et al., 2025); we leave multi-GPU execution to future work.

5.1 Microbenchmarks

Matrix multiplications. Table 2 compares the six matrix multiplications of one BERT-Base encoder layer in ELLMo, THOR (Moon et al., 2025), Powerformer (Park et al., 2025), and Tricycle (Lim et al., 2025). We compare Key-Switch counts because Key-Switch is the dominant primitive in CKKS, and we derive the Tricycle FFN counts from the complexity analysis in Section 6.3 of Lim et al. (2025). ELLMo issues about as many Key-Switch operations as THOR, more than Powerformer, and fewer than Tricycle, yet it is 24×, 27×, and 7× faster, respectively. The count alone does not explain this: the latency of a linear layer also depends on hoisting, multiplicative depth, and the data layout in GPU kernels, and ELLMo improves all three by hoisting across output tiles, running SPCMM at a low depth, and fusing the GPU kernels. Appendix A.5.2 isolates the gain of each technique as an ablation study.

Weight memory. SPCMM encodes each weight at a low level, so a weight plaintext carries few limbs. With ELLMo, the stored weights take 5.0 GB for BERT-Base and GPT-2, 27.5 GB for Llama-3.2-1B, 190.6 GB for Llama-3-8B classification / prefill, and 154.6 GB for Llama-3-8B decoding. This is 3.2× less than CERIUM for BERT-Base (16 GB) and 5.2× less for Llama-3-8B (982 GB) (Jayashankar et al., 2025), and 2× less than Cachemir (694 GB) when the prefill and decode weights of Llama-3-8B are stored together (345.2 GB). BERT-Base fits in GPU memory together with its keys and ciphertexts. For both Llama models, the weights stay in host memory (256 GB of RAM, Intel Xeon Gold 6326) and are streamed layer by layer over PCIe, overlapped with the compute of the previous layer; the reported latencies include any overheads due to these transfers.

Bootstrapping. L_{eff}, the number of levels available after a bootstrap, is a trade-off. A larger L_{eff} needs fewer bootstraps, but every ciphertext and key takes more memory, operations run on more limbs, and each bootstrap becomes slower (Ebel et al., 2025). ELLMo uses $L_{\text{eff}} = 10$ to keep memory low and runs BERT-Base on SST-2 with 439 bootstrapped ciphertexts. ATLAS reports 379 to 399 bootstrapped ciphertexts on SST-2 with no accuracy loss, at $L_{\text{eff}} = 14$ and with PCMMs

Table 3: End-to-end encrypted classification against the plaintext model. Accuracy for SST-2, RTE, and QNLI, and F1 for MRPC, all measured under encryption and averaged over three runs. Latency is for one unamortized 128-token input on an A100 with no model change.

	BERT-Base				Llama-3.2-1B		Llama-3-8B	
	SST-2	MRPC	RTE	QNLI	SST-2	MRPC	SST-2	MRPC
Plaintext	92.32	0.9026	69.68	90.99	95.18	0.8754	93.46	0.8274
Encrypted	92.43	0.9026	69.55	91.05	95.18	0.8769	93.42	0.8249
Latency (s)	50.23	50.96	57.42	56.88	189.2	204.1	624.8	611.43

Table 4: BERT-Base inference time for one 128-token input on a single A100 80GB, and accuracy difference with plaintext, in the notation of Al Badawi et al. (2026): ○ $\leq 0.5\%$; • 0.5 to 1.5%; ▲ $> 1.5\%$ or requires retraining; − not reported for a dataset. ELLMo latency is a range over the SST-2, MRPC, RTE, and QNLI tasks. † denotes amortized latency over a batched execution.

	THOR	Powerformer	MOAI	Euston	Tricycle	CERIUM	STIP	ELLMo
Accuracy	○	▲	○	○	−	○	▲	○
Latency (s)	602.26	344.52	283.95†	103.1†	100.55	66.00	64.23†	**50.2–57.4**

allowed at higher levels. At the same L_{eff}, the ELLMo calibrator bootstraps 379 ciphertexts even with every PCMM pinned to the lowest levels. Placing PCMMs at higher levels would lower this count further, but we keep them pinned so that weight plaintexts have a small memory footprint.

5.2 Classification

We now compose the ELLMo-optimized kernels into full end-to-end encrypted inference. We first confirm that the encrypted model achieves accuracy comparable to the plaintext model, then compare latency and accuracy against prior work.

Accuracy. Table 3 reports the encrypted model against the plaintext model on the same inputs. Encrypted accuracy with ELLMo is within 0.3% of the plaintext model on every task. These numbers come from the calibrator alone: we do not modify the model, we keep the original nonlinear functions, and we do not do fine-tuning. The latency of a model differs across datasets because the calibrator adapts to their input ranges, which changes the polynomial degrees and the bootstrap counts (Section 4).

End-to-end comparison on BERT-Base. Table 4 compares ELLMo with prior work, using the accuracy notation of the FHE-LLM SoK (Al Badawi et al., 2026). MOAI (Zhang et al., 2026), Euston (Gao et al., 2026), and STIP (Wang et al., 2026) amortize the latency over a batch, and Powerformer (Park et al., 2025) and STIP modify the nonlinearities, while ELLMo reports unamortized latency with no model change or fine-tuning. Prior works use one approximation and bootstrapping placement for all datasets, so their latency does not vary across them. In ELLMo, the matrix multiplications take the same time on every dataset, but the calibrator adapts the nonlinear approximations and the bootstrap count to the input ranges of each task, so we report its full range over the four tasks. Even on its slowest task, RTE at 57.4 s, ELLMo is $10.5\times$ faster than THOR, $6.0\times$ faster than Powerformer, $4.9\times$ faster than MOAI, and $1.75\times$ faster than Tricycle. On SST-2, matrix multiplications take 8.6 s, 3.6 s in the four SPCMMs and 5.0 s in the two CCMMs, and bootstrapping takes 30.1 s, 60% of the total (Table 10).

Powerformer replaces the Softmax with a fine-tuned surrogate, while ELLMo keeps the original functions and weights. As a result, Softmax takes 4.93 s in ELLMo against 0.75 s for Powerformer. Such techniques are orthogonal to ELLMo and would save about 4.2 s, at the cost of fine-tuning and accuracy. Compared to THOR, which also keeps the original nonlinearities, the Softmax, LayerNorm, and GELU of ELLMo together take 9.0 s instead of 56.2 s, and its bootstrapping takes 30.1 s instead of 337.9 s. Among compiler frameworks, CERIUM (Jayashankar et al., 2025) reports the best unamortized BERT-Base latency, 66 s on RTE, where ELLMo takes 57.4 s, and SPCMM and the calibrator are compatible with such frameworks. NEXUS (Zhang et al., 2025) reports the latency

Table 5: Encrypted token perplexity and latency on WikiText with 128-token windows, measured during teacher-forced prefill.

	GPT-2	Llama-3.2-1B	Llama-3-8B
Plaintext (PPL)	63.49	26.20	16.95
Encrypted (PPL)	63.39	26.24	17.01
Latency (s)	68.7	211.6	633.7

Table 6: Llama-3-8B latency with ELLMo. Decode: latency of one decoding step, time per output token (TPOT): Decode + amortized prefill.

(input, context)	(128, 256)	(64, 512)	(16, 1024)
Prefill (s)	633.7	433.0	284.2
Decode (s)	74.4	100.1	152.9
TPOT (s)	79.4	101.1	153.2

of individual building blocks rather than of an end-to-end inference, and its layers need repacking operations between them, so we exclude it from the latency comparison.

End-to-end results on Llama models. ELLMo runs Llama-3.2-1B and Llama-3-8B with no model change or fine-tuning, in 189 to 204 s and 611 to 625 s per input, respectively (Table 3). Encrypted accuracy is within 0.3% of the plaintext model. Euston (Gao et al., 2026) and STIP (Wang et al., 2026) evaluate a Llama-3-8B-sized layer with LayerNorm and a two-matrix FFN instead of the original RMSNorm and SwiGLU blocks, so their latencies are not directly comparable.

5.3 Generation

Generation stresses FHE differently from classification: decoding produces one token at a time, so the sequence length collapses to $T = 1$, and the key-value cache is kept and grown under encryption.

Perplexity. Table 5 reports perplexity with teacher forcing on 128-token windows of the WikiText test set, so the encrypted and plaintext models score the same tokens. Encrypted perplexity is at most 0.35% higher than plaintext, on Llama-3-8B (17.01 against 16.95). Cachemir (Yu et al., 2026b) and ROSETTA (Yu et al., 2026a) instead report perplexity on a plaintext model with approximated nonlinear functions, which leaves out the error that an end-to-end encrypted inference incurs. In this setup, and with different windows, ROSETTA reports a 2.0% increase on Llama-3-8B for both its own method and Cachemir. The latency of one window grows about $3\times$ from GPT-2 to Llama-3.2-1B and again to Llama-3-8B, while the parameter count grows $10\times$ and $6.5\times$.

Prefill and decoding. Table 6 reports Llama-3-8B latency at context lengths of 256, 512, and 1024. Decoding keeps the tile layout and kernels of prefill for attention, runs the feed-forward projections as matrix-vector products on the single live row, and appends each new key and value to the encrypted cache with a masked addition; the calibrator produces a separate plan for decode-time ranges (Appendix A.3.3). A $4\times$ longer context costs only about $2\times$ more per decoding step, because the linear layers do not depend on the context length. For Llama-3-8B, CERIUM reports 675 s prefill and Cachemir reports 3.02 s per layer per output token, which gives 97 s for the 32 layers if there is no overhead between layers, at a 512-token context. ROSETTA reports 310.8 s per Llama-3-8B token at a 512-token context. At this context, ELLMo takes 101.1 s per Llama-3-8B token, which is comparable to Cachemir and $3.1\times$ faster than ROSETTA.

6 Conclusion

ELLMo targets the two main costs of encrypted transformer inference: matrix multiplication and bootstrapping. SPCMM reduces the rotations, depth, and weight memory of plaintext-ciphertext products. The calibrator selects nonlinear approximations that minimize bootstrapped ciphertexts, with a convergence guarantee for inverse square roots. Both keep the model unchanged and can be ported to other CKKS libraries and compiler frameworks. On a single A100, BERT-Base inference takes 50.2 to 57.4 s, faster than all prior work, including the compiler-based CERIUM. Llama-3-8B takes up to 625 s for classification, 634 s for prefill, and 101 s per generated token at a 512-token context, with accuracy and teacher-forced perplexity within 0.35% of the plaintext model.

USE OF AI

We used AI tools to edit the text for readability and consistency, for literature search, to draft figure and plotting scripts, and as a coding assistant; every suggestion was reviewed by the authors. All ideas, algorithms, proofs, experiments, and conclusions are the work of the authors.

REPRODUCIBILITY STATEMENT

Section 5 lists the CKKS parameters, the security level, the hardware, and the models and datasets. The calibration examples are randomly sampled from training splits, with WikiText cut into 128-token windows. The proofs of the propositions and lemmas are given in Appendix A.3. An anonymized repository will be made available to the reviewers during the review period, and the source code will be released publicly upon publication.

REFERENCES

Rashmi Agrawal and Ajay Joshi. *On Architecting Fully Homomorphic Encryption-based Computing Systems*. Springer, 2023. ISBN 978-3-031-31753-8. doi: 10.1007/978-3-031-31754-5.

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 *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 1–3, Ghent, Belgium, 2025. IEEE. URL https://github.com/CAPS-UMU/FIDESlib. Poster paper.

Hirotugu Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974.

Ahmad Al Badawi, Andreea Alexandru, Yuriy Polyakov, and Vinod Vaikuntanathan. SoK: Private LLM inference using approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2026/935, 2026. URL https://eprint.iacr.org/2026/935.

Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 587–617. Springer, 2021.

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. Cryptology ePrint Archive, Paper 2024/463, 2024. URL https://eprint.iacr.org/2024/463.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pp. 409–437. Springer, 2017.

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*, pp. 360–384. Springer, 2018.

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*, pp. 347–368. Springer, 2019.

Wonhee Cho, Guillaume Hanrot, Taeseong Kim, Minje Park, and Damien Stehlé. Fast and accurate homomorphic softmax evaluation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 4391–4404, 2024.

Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer Series in Statistics. Springer, 2001.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Third international workshop on paraphrasing (IWP2005)*, 2005.

Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, pp. 734–749, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716008. URL https://doi.org/10.1145/3676641.3716008.

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, Paper 2026/046, 2026. URL https://eprint.iacr.org/2026/046.

Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and William B Dolan. The third pascal recognizing textual entailment challenge. In *Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing*, pp. 1–9, 2007.

Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pp. 201–210. PMLR, 2016.

Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. In *Annual International Cryptology Conference*, pp. 93–120. Springer, 2018.

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.

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*, pp. 1209–1222. Association for Computing Machinery, 2018. URL https://doi.org/10.1145/3243734.3243837.

Jongmin Kim, Wonseok Choi, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library for CUDA GPUs. *arXiv preprint arXiv:2407.13055*, 2024. URL https://doi.org/10.48550/arXiv.2407.13055.

Lawrence Lim, Vikas Kalagi, Divyakant Agrawal, and Amr El Abbadi. Tricycle: Private transformer inference with tricyclic encodings. Cryptology ePrint Archive, Paper 2025/1200, 2025. URL https://eprint.iacr.org/2025/1200.

Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Chiara Marcolla, Victor Sucasas, Marc Manzano, Riccardo Bassoli, Frank H.P. Fitzek, and Najwa Aaraj. Survey on fully homomorphic encryption, theory, and applications. Cryptology ePrint Archive, Paper 2022/1602, 2022. URL https://eprint.iacr.org/2022/1602.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017.

Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. THOR: Secure transformer inference with homomorphic encryption. Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, 2025. URL https://eprint.iacr.org/2024/1881.

Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 2006.

Dongjin Park, Eunsang Lee, and Joon-Woo Lee. Powerformer: Efficient and high-accuracy privacy-preserving language model with homomorphic encryption. *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 11090–11111, 2025. doi: 10.18653/v1/2025.acl-long.543. URL https://aclanthology.org/2025.acl-long.543/.

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 FHE-based privacy-preserving ML: Higher throughput, longer inputs for LLama-3-8B, 2026. URL https://arxiv.org/abs/2601.18511.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. OpenAI Technical Report, 2019.

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 (ICLR)*, 2025.

Rostin Shokri and Nektarios Georgios Tsoutsos. Terrazzo: Memory-aware GPU framework for private transformer inference. Cryptology ePrint Archive, Paper 2026/1870, 2026. URL https://eprint.iacr.org/2026/1870.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1631–1642, 2013.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations (ICLR)*, 2019.

Zihao Wang, Rongmao Chen, Xinwen Gao, Yi Wang, Lin Liu, Zixin Lan, Zhaoyu Wang, Shaojing Fu, Qiong Wang, and Xinyi Huang. STIP: Efficient and secure non-interactive transformer inference via compact packing. Cryptology ePrint Archive, Paper 2026/174, 2026.

Jianhang Xie, Sicheng Tan, Vishnu Naresh Boddeti, and Zhichao Lu. ATLAS: Automated approximation of transformers for efficient homomorphic inference in one hour. Cryptology ePrint Archive, Paper 2026/1817, 2026. URL https://eprint.iacr.org/2026/1817.

Jiangrui Yu, Baosheng Zhang, Liang Kong, Lin Ding, Yi Chen, Ye Yu, Mingzhe Zhang, and Meng Li. ROSETTA: Efficient and accurate privacy-preserving LLM decoding via hybrid CKKS/TFHE evaluation. Cryptology ePrint Archive, Paper 2026/2047, 2026a. URL https://eprint.iacr.org/2026/2047.

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

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

Linru Zhang, Xiangning Wang, Jun Jie Sim, Zhicong Huang, Jiahao Zhong, Huaxiong Wang, Pu Duan, and Kwok Yan Lam. Moai: Module-optimizing architecture for non-interactive secure transformer inference. In *International Conference on Learning Representations*, 2026.

A Appendix

A.1 Fully Homomorphic Encryption

This appendix introduces the parts of CKKS (Cheon et al., 2017) that our contributions build on, and Agrawal & Joshi (2023) gives a detailed treatment.

Encoding and slots. CKKS encodes a vector of $n = N/2$ complex numbers, called slots, into one polynomial of degree N, after scaling the values by Δ and rounding them to integers. Additions and multiplications of polynomials then act slot by slot on the vectors. Polynomials are stored in the Number Theoretic Transform (NTT) domain, where a product of two polynomials is an element-wise product of their evaluations, which Section 3.1 relies on.

Levels and rescaling. The modulus is a product of primes $Q = q_0 \cdot q_1 \cdots q_L$, and each coefficient is stored as its residues modulo these primes, one limb per prime (Cheon et al., 2019). A product of two encoded values carries the scale Δ^2, and $\mathrm{Rescale}$ divides by the last prime $q_\ell \approx \Delta$, which restores the scale and drops one limb. So every multiplication consumes one level, and a ciphertext at level ℓ has $\ell + 1$ limbs.

Key switching. A ciphertext-ciphertext product, a Rotate, and a complex conjugation leave the result encrypted under a different key. $\mathrm{Key\text{-}Switch}$ restores the original key with a public evaluation key: it raises the ciphertext to a larger modulus (ModUp), multiplies it by the key, and brings it back down ($\mathrm{ModDown}$). The two modulus conversions dominate the cost, and Appendix A.2.2 shows how hoisting amortizes them.

Bootstrapping. A ciphertext at level 0 can no longer be multiplied. $\mathrm{Bootstrap}$ (Cheon et al., 2018) evaluates the decryption homomorphically and returns the same message with L_{eff} levels, after consuming $L - L_{\text{eff}}$ levels itself. We use $N = 2^{16}$, $L = 26$, and $L_{\text{eff}} = 10$, which give 128-bit security (Bossuat et al., 2024).

A.2 Transformer Preliminaries

A.2.1 Transformers

Attention. Each head computes

$$\mathrm{Attn}_h = \mathrm{Softmax}\left(\frac{Q_h K_h^\top}{\sqrt{d_h}} + M\right) V_h, \tag{4}$$

where Softmax acts on each row of the $T \times T$ score matrix and $d_h = D/H$ (Vaswani et al., 2017). In BERT and GPT-2, $W_Q, W_K, W_V \in \mathbb{R}^{D \times D}$ (Devlin et al., 2019; Radford et al., 2019). Llama uses grouped-query attention with $H_{kv} = 8$ key-value heads, so $W_K, W_V \in \mathbb{R}^{D \times H_{kv} d_h}$, and each key-value head is shared by H/H_{kv} query heads (Llama Team, AI @ Meta, 2024). Llama applies rotary position embeddings (RoPE) to Q and K, which rotate each pair of columns by an angle that depends on the token position, while GPT-2 adds learned position embeddings to the input. In BERT, M masks only the padding tokens. In GPT-2 and Llama, M is $-\infty$ above the diagonal, so each token attends only to itself and earlier tokens.

Feed-forward network. The FFN of BERT and GPT-2 is $\mathrm{GELU}(XW_1)W_2$ with $W_1 \in \mathbb{R}^{D \times 4D}$ and $W_2 \in \mathbb{R}^{4D \times D}$. Llama uses $(\mathrm{SiLU}(XW_{\text{gate}}) \odot XW_{\text{up}})\, W_{\text{down}}$, with $D_{ff} = 8192$ in Llama-3.2-1B and 14336 in Llama-3-8B. The gate adds one Hadamard product between two ciphertexts per layer.

Normalization. LayerNorm subtracts the mean of each token, divides by its standard deviation, and applies a learned scale and bias. BERT applies it after each sublayer, and GPT-2 before each sublayer. Llama applies RMSNorm before each sublayer, which divides each token by its root mean square without subtracting the mean. All three need an inverse square root.

A.2.2 Encrypted Transformer Inference

Threat model. The client encrypts its input with CKKS and sends it to a semi-honest server that holds the model weights in plaintext. The privacy of the input follows from the semantic security of CKKS. The confidentiality of the weights, leakage through the decrypted output, and side channels are out of scope.

Packing. A matrix is split into tiles, and each tile is packed row-major into the $n = 32768$ slots of one ciphertext. Section 3 uses square $d \times d$ tiles for readability, while our implementation uses 128×256 tiles (Appendix A.3.2). For example, a BERT-Base activation of size 128×768 takes three tiles.

Encrypted matrix multiplication. The textbook product $(A \times B)_{i,j} = \sum_k A_{i,k} B_{k,j}$ indexes individual entries, and CKKS can only move entries between slots with rotations. The JKLS method (Jiang et al., 2018) instead uses transforms that permute the entries so that the entries to be multiplied land in the same slot. Each transform is a sum of rotated copies of the input multiplied by plaintext masks, and consumes one level. A transform with diagonals $m_0, \ldots, m_{d-1}$ computes $\sum_k m_k \odot \mathrm{Rotate}_k(A)$, which costs $O(\sqrt{d})$ rotations with the baby-step giant-step (BSGS) decomposition (Halevi & Shoup, 2018). With double hoisting (Bossuat et al., 2021), the baby steps rotate the same ciphertext and share one ModUp, and each inner sum needs one $\mathrm{ModDown}$ after the addition.

Nonlinear functions. Every nonlinear function is replaced by a polynomial that approximates it on a fit interval $[a, b]$. A Chebyshev polynomial of degree δ consumes about $\lceil \log_2(\delta + 1) \rceil$ levels. Outside $[a, b]$, the error grows quickly, and later sums over a row, such as the Softmax denominator, spread it to the whole row, so the interval must cover every input the model may see. GELU and SiLU are approximated directly. Softmax needs an exponential and a reciprocal, and prior work subtracts a calibrated constant per layer instead of the costly row maximum (Zhang et al., 2025). The iterative Softmax of Cho et al. (2024) evaluates $e^{x/2^k}$ on a narrow interval and repeats a normalize-and-square step k times, which extends the valid input range to about $[-2^k \ln n, 0]$. The reciprocal and the inverse square root refine a polynomial seed y_0 with Newton-Raphson (NR) iterations, $y_{k+1} = y_k\,(2 - x\,y_k)$ and $y_{k+1} = \frac{1}{2}\,y_k\,(3 - x\,y_k^2)$, respectively. NR converges quickly when the seed is close to the result but diverges otherwise, which Section 4 addresses.

Bootstrapping cost. A bootstrap acts on one ciphertext, so its total cost grows with the number of ciphertexts that are live when levels run out. This number varies widely across a transformer: a Llama-3-8B activation spans 3.5× more tiles after the FFN up projection than before it. This is why the calibrator of Section 4 counts bootstrapped ciphertexts rather than bootstrap calls or depth.

A.3 Details of SPCMM

A.3.1 Proofs of Proposition and Lemmas

Proof of Proposition 3.1. Fix $i, j \in \mathbb{Z}_d$. By the definitions of ϕ, τ, and ζ, the (i, j) entry of the sum is

$$\sum_{k=0}^{d-1} \phi^k(A)_{i,j} \cdot \big(\zeta^k \circ \tau(B)\big)_{i,j} = \sum_{k=0}^{d-1} A_{i,\,j+k} \cdot \tau(B)_{k,j} = \sum_{k=0}^{d-1} A_{i,\,j+k} \cdot B_{j+k,\,j}. \tag{5}$$

As k ranges over $[0, d)$, the index $t = j + k \pmod d$ takes every value exactly once, so the sum equals $\sum_t A_{i,t}\, B_{t,j} = (AB)_{i,j}$. □

Plaintext representation. Encode maps a slot vector $z \in \mathbb{C}^n$, with $n = N/2$, to a polynomial m of degree less than N with integer coefficients, through a scaled inverse discrete Fourier transform followed by rounding. Each RNS limb of m is stored in the NTT domain (Agulló-Domingo et al., 2025), that is, as the N values of m at the N evaluation points of the NTT, in bit-reversed order. Multiplication of polynomials is entrywise in this domain and decodes to the slot-wise product.

Proof of Lemma 3.1. The encoding and the NTT are both Fourier transforms, and a Fourier transform turns repetition into sparsity and sparsity into repetition. The proof applies this twice and then accounts for the storage order.

(1) Slots to coefficients. A c-periodic z is a c-slot vector repeated $d = n/c$ times. Its encoding therefore has a nonzero coefficient only at every d-th position, that is, $m(X) = \hat{u}(X^d)$ with $\deg \hat{u} < 2c$, where $\hat{u}$ is the encoding of the c-slot vector on its own. This is the sparse packing of CKKS (Cheon et al., 2018).

(2) Coefficients to NTT values. The NTT evaluates m at N points, indexed $t = 0, \ldots, N-1$ in natural order. Since m depends only on X^d, and the d-th powers of the evaluation points repeat with period $N/d = 2c$, the value at index t depends only on $t \bmod 2c$. So a limb holds $2c$ distinct values, each d times.

(3) Storage order. The limb is stored in bit-reversed order, and the value at natural index t depends only on the low $\log_2(2c)$ bits of t. Bit reversal moves these bits to the high bits of the position, so equal values occupy d consecutive positions, and the limb is $2c$ runs of length d. □

Proof of Lemma 3.2. Multiplication in R_q is entrywise in the NTT domain, and by Lemma 3.1 the value of B is the same at every position of a run. So multiplying each run of A by the one stored value computes the same N products per limb as the dense entrywise product, which decodes to $A \odot B$. □

Proof of Proposition 3.2. Every row of $\zeta^k \circ \tau(B)$ is the same, so its slot vector is c-periodic and Lemma 3.2 applies to every term of Proposition 3.1. □

A.3.2 Rectangular Tiles

In our deployed SPCMM implementation, the tiles are rectangular: a ciphertext holds a tile $A \in \mathbb{R}^{d_1 \times d_2}$ of the activation, and the plaintext holds a tile $B \in \mathbb{R}^{d_2 \times d_1}$ of the weight, stored as $B^\top$ so that both tiles are $d_1 \times d_2$, with d_1 dividing d_2 and $R = d_2/d_1$. The product $A \times B$ is $d_1 \times d_1$, and it appears in the output tile repeated R times across the d_2 columns. A mask then keeps one copy, so R weight tiles share one output ciphertext, one per column block. The deployed configuration uses 128×256 tiles, resulting in $R = 2$.

Proposition A.1 (ShallowPCMM, rectangular tiles)**.** Use $\phi^k(X)_{i,j} = X_{i,\, j+k}$ with the column index taken modulo d_2. Let $\tau(B)$ be the $d_1 \times d_2$ matrix $\tau(B)_{k,j} = B_{(k+j) \bmod d_2,\ j \bmod d_1}$, and let $\zeta^k(X)$ be the $d_1 \times d_2$ matrix whose every row is row k of X. Write S for the d_1-term sum

$$S = \sum_{k=0}^{d_1-1} \phi^k(A) \odot \big(\zeta^k \circ \tau(B)\big). \tag{6}$$

Then, for every $i \in [d_1]$ and $j \in [d_2]$,

$$\sum_{l=0}^{R-1} S_{i,\,(j+l\,d_1) \bmod d_2} = (A \times B)_{i,\, j \bmod d_1}. \tag{7}$$

Proof. Fix $i \in [d_1]$ and $j \in [d_2]$. By the definitions of ϕ, τ, and ζ,

$$S_{i,j} = \sum_{k=0}^{d_1-1} A_{i,\, j+k} \cdot B_{(j+k) \bmod d_2,\ j \bmod d_1}, \tag{8}$$

a partial contraction over the d_1 consecutive indices $t = j + k \pmod{d_2}$. Now sum over the R column blocks. Since $(j+l\,d_1) \bmod d_1 = j \bmod d_1$, every term of equation 7 pairs A with the same column $j \bmod d_1$ of B, and as (l, k) ranges over $[R] \times [d_1]$ the index $t = j + l\,d_1 + k \pmod{d_2}$ takes every value in $[d_2]$ exactly once. Hence the left side equals $\sum_t A_{i,t}\, B_{t,\, j \bmod d_1} = (AB)_{i,\, j \bmod d_1}$. For square tiles $R = 1$, the outer sum has the single term $l = 0$ and S is already the product, which is Proposition 3.1. □

The sum equation 6 has d_1 terms, so the BSGS decomposition uses $b\,g = d_1$, the fold equation 7 adds $\log_2 R$ rotations per output tile, and the second rotation of each column shift is shared across all k. Lemmas 3.1 and 3.2 apply with $c = n/d_1$ slots per row and runs of length d_1, so each of the d_1 sparse plaintexts stores N/d_1 values per limb and all of them together take exactly N, one dense encoding of B, for any d_1 and d_2. The level count is unchanged: two multiplicative levels with lazy rescaling, with column-shift masks folded into the plaintexts.

A.3.3 Decoding Layout

During decoding, the new token's activation is a $1 \times D$ row that stays in the tile layout of prefill: it occupies row $t = (T-1) \bmod d_1$ of its tile, and the other rows hold stale values from earlier steps. A row-t plaintext selector is applied at every site that consumes the row, so the stale rows never reach the output. The attention and output projections run CCMMs. The weight projections instead gather the row into one ciphertext, with entry w at slot w, and run as baby-step giant-step matrix-vector products, since a tile product would spend $d_1 - 1$ rows on stale values. The attention scores multiply the query against the cached keys with the CCMM of Section 3.2, and a context of C tokens occupies $\lceil C/d_1 \rceil$ cache blocks, so only the attention cost grows with the context length.

The cache holds K and V as ciphertexts at the fixed CCMM level. The new key and value already sit at row t of their own tiles, so appending them is a masked addition: the step applies the row-t selector, which costs one level, and adds the result into the cache tile, which keeps the cache from accumulating depth across steps. Decode-time activations have different ranges from prefill, so the calibrator runs once more over them and emits a separate plan for decoding.

A.4 Calibration

A.4.1 Search space

The calibration set is a subset of the training data, 1000 examples by default, and the precision sweep tries every p in a configurable set, $\{4, 8, 12, 16, 20\}$ by default. For each site the planner enumerates five polynomial degrees (5, 13, 27, 59, and 119) and 1 to 10 NR iterations, chosen for our CKKS parameters, which is 50 candidates; a softmax site also enumerates the iteration constant $k \le 4$ (Cho et al., 2024), giving 200. Llama-3-8B has up to 8 nonlinear operators in each of its 32 layers, so up to 256 sites, one softmax per layer, and coupling them through a single end-to-end tolerance would make that a joint space of $200^{32} \times 50^{224} \approx 10^{454}$ plans. The precision sweep decouples the sites, so each is checked on its own against the precision target and the convergence test, and $32 \times 200 + 224 \times 50 = 17{,}600$ candidates remain.

A.4.2 Tail Estimation

A fit interval must cover every input the deployed model produces, but the calibration set is finite, so its observed minimum and maximum are not that interval. An input just past the edge makes the Chebyshev polynomial diverge, and the row sums downstream spread the error over the whole row. The usual fix is to pad the observed range by a hand-set factor, which is a guess in both directions: too little lets an input escape, and too much wastes polynomial degree on a range that never occurs. ELLMo instead estimates how far the tail extends.

Fitting the tail. The plaintext pass records the minimum and maximum of every site for each calibration example. Such per-example extremes follow the generalized extreme value (GEV) distribution (Coles, 2001), whose shape parameter says whether the tail is bounded. ELLMo fits this distribution and places each edge at the value that one inference exceeds with probability 10^{-6}, and never inside the observed range. The shape matters. A Gumbel fit assumes an unbounded exponential tail, but every quantity we calibrate is a bounded function of bounded inputs, so ELLMo fits a bounded tail (Akaike, 1974). On the BERT-Base RTE pooler, the Gumbel fit asks for an edge at 142.5 where the bounded fit finds a ceiling at 51.7, and covering the extra range would cost $29\times$ the $\tanh$ error at the same degree. Heavy tails with no ceiling are excluded, because their estimates explode: one LayerNorm variance of BERT-Base SST-2 would get an edge at 1.6×10^5 against an observed maximum of 5.157. Lower edges are fitted the same way, except that a variance feeding an inverse square root is fitted in log space so that its edge stays positive.

A.4.3 PLANNER

For every surviving candidate a level simulation places a bootstrap wherever the level budget runs out, and the planner keeps the candidate with the fewest bootstrapped ciphertexts, $\sum_b n_b$, where n_b is the number of ciphertexts refreshed at bootstrap b.

A.4.4 THE NR-CHEBYSHEV-NR SANDWICH FOR INVERSE SQUARE ROOT

LayerNorm and RMSNorm need $x^{-1/2}$ on a calibrated interval $[\mathrm{lo}, \mathrm{hi}]$ with $\mathrm{lo} > 0$. The cheap way to compute it is a Chebyshev polynomial seed refined by Newton-Raphson (NR) iterations (Nocedal & Wright, 2006). The catch is that NR is safe only from a seed close enough to the answer. From an undershoot of any size, and from an overshoot below $(\sqrt{17} - 3)/2 \approx 0.56$, the relative error shrinks at every step; from an overshoot between 0.56 and $\sqrt{3} - 1 \approx 0.73$ the first step lands in a large undershoot before converging; and above 0.73 the iterate changes sign and the ciphertext is corrupted. We call the region where the error shrinks at every step the convergence region, and require every seed to lie in it with a margin, so that CKKS noise cannot push an iterate near 0. A Chebyshev polynomial fitted on a wide interval cannot guarantee such a seed, so ELLMo runs k_1 NR iterations before it.

The pre-NR stage starts from a constant chosen from lo and hi so that the overshoot stays below the limit for every x in the interval. Its purpose is range compression. Each iteration moves the estimate y toward $x^{-1/2}$, so the normalized value $x\,y^2$, which equals 1 when the estimate is exact, is pulled into a window around 1 that tightens with every iteration, however wide $[\mathrm{lo}, \mathrm{hi}]$ is. The Chebyshev polynomial then approximates $t^{-1/2}$ on that window only, which the calibrator measures on a grid and pads for CKKS noise. Multiplying its output by the pre-NR estimate gives the estimate that the k_2 closing NR iterations refine.

A candidate is accepted only if the relative error of that estimate stays inside the convergence region with a margin at every grid point, so the closing iterations converge by construction and their error squares at each step. The calibrator sets k_2 to the first count that meets the site's precision target, rejects any window whose left edge approaches 0, where $t^{-1/2}$ is singular, and charges the pre-NR stage $2k_1 + 2$ levels over a plain Chebyshev seed. The reciprocal needs no sandwich, because its NR iteration converges from any seed with relative error below 1, and a Chebyshev seed meets this at every reciprocal site in our models.

A.5 IMPLEMENTATION DETAILS AND ADDITIONAL RESULTS

A.5.1 CHECKPOINTS, DATASETS, AND EVALUATION SIZES

Table 7: Checkpoints used for the reported results, with the first eight characters of the pinned Hugging Face revision and the number of evaluated examples. WikiText-2 counts are 128-token windows, of which 127 tokens are scored.

Model	Task	Checkpoint	Revision	Examples
BERT-Base	QNLI	`gchhablani/bert-base-cased-finetuned-qnli`	`e6c6176e`	5463
BERT-Base	SST-2	`gchhablani/bert-base-cased-finetuned-sst2`	`e3a2a13e`	872
BERT-Base	MRPC	`gchhablani/bert-base-cased-finetuned-mrpc`	`cdece369`	408
BERT-Base	RTE	`google-bert/bert-base-uncased`, finetuned	—	277
GPT-2 Small	WikiText-2	`gpt2`	`607a30d7`	512
Llama-3.2-1B	SST-2	`emirhanboge/LLaMA_1B_sst2_FullFT`	`d0b3f1e7`	872
Llama-3.2-1B	MRPC	`jhj1769/Llama1B_mrpc`	`f8db4450`	408
Llama-3.2-1B	WikiText-2	`meta-llama/Llama-3.2-1B`	`4e20de36`	512
Llama-3-8B	SST-2	`skm2003/llama3-8b-glue-finetuned`, folder sst-2	`dc2baa7f`	872
Llama-3-8B	MRPC	`skm2003/llama3-8b-glue-finetuned`, folder mrpc	`dc2baa7f`	408
Llama-3-8B	WikiText-2	`meta-llama/Llama-3.1-8B`	`d04e592b`	512

Table 7 lists the checkpoint and the number of evaluated examples behind every number in Tables 3 and 5. The Llama classification checkpoints are full fine-tunes with a sequence-classification head, a linear layer on the hidden state of the last token, which runs as one more PCMM after the final RMSNorm. The BERT checkpoints keep the standard pooler and classification head.

The BERT-Base RTE head is our own plaintext fine-tune of `google-bert/bert-base-uncased`. CERIUM (Jayashankar et al., 2025) reports BERT-Base accuracy only on RTE and only for its own fine-tune of this base model, so we fine-tune the same base model to get a matching comparison point. This is ordinary plaintext task training and is unrelated to encryption: we train for 10 epochs at learning rate 2×10^{-5}, batch size 16, maximum length 256, and seed 42, and keep the epoch with the best validation accuracy, 69.7% in plaintext.

The datasets are the GLUE validation splits of SST-2 (872 examples), MRPC (408), RTE (277), and QNLI (5,463), and the test split of `wikitext-2-raw-v1`. Under encryption we evaluate every example of the classification splits, and 512 non-overlapping WikiText windows of 128 tokens, each a start-of-text token followed by 127 text tokens scored with teacher forcing. Examples longer than one 128-token block, which occur in RTE and QNLI, run on a separate implementation that supports multiple token blocks. These examples count toward the reported accuracy, but the latencies in Table 3 are measured on the single-block path with one 128-token input. Every reported number is the mean of three independent encryptions of the whole split.

A.5.2 Ablations

Tables 8 and 9 add the optimizations of Section 3 step by step, for one BERT-Base layer. Complex packing (Section 3.3) halves the key switches of $\times W_{FFN2}$ and cuts those of the other PCMMs by 1.5×. SPCMM removes the transform of the ciphertext, which cuts the key switches of $\times W_O$ by 2.0× and of $\times W_{FFN2}$ by 3.9×, and the latency of all four PCMMs by 1.6× to 2.6×. The sparse encoding leaves the key switches unchanged, but reading the compact plaintexts cuts the latency by 1.3× to 1.6×. Hoisting across output tiles gives the largest cut: every weight tile in a row of the product multiplies the same rotated copies of the input, so they share one ModUp, and together with scale folding this cuts the key switches by 3.5× to 14.2×. Kernel batching and fusion leave the key switches unchanged, and cut the latency by 2.3× to 3.4×, by launching the products of a row in one kernel and keeping intermediate values out of global memory. For the CCMMs, imaginary packing and the head fold (Section 3.2) cut the key switches of $Q \times K^\top$ by 2.0× and of $\times V$ by 2.7×, double hoisting cuts those of $Q \times K^\top$ by a further 1.4×, and kernel fusion cuts the latency by 1.5× to 1.9×.

GPU kernels. On the GPU, a ciphertext at level ℓ takes $(\ell + 1)$ MB for $N = 2^{16}$, so every intermediate result that passes through global memory is costly. An output tile is the sum of several tile products, and without fusion, each product runs as its own kernel. We fuse the sum over the tile pairs and over the baby steps of each tile product into one kernel, which reads the inputs and writes the output once. The sparse plaintexts are read in compact form, further reducing memory traffic.

Table 8: Ablation of the PCMM optimizations for one BERT-Base layer, with 128×256 tiles on an A100 80GB. Each row adds the optimizations it names to the row above. Each cell gives the number of key switches / the latency in ms, where each product is timed in isolation on operands at level $\ell = 2$, as the minimum of 10 runs after 3 warm-up runs; the in-model times in Table 2 include the surrounding packing and rescaling, so they differ from the last row. The baby-step size b is 32 for $\times W_{Q/K/V}$ and $\times W_{FFN1}$, 16 for $\times W_O$, and 8 for $\times W_{FFN2}$. *The first two rows have no fused $Q/K/V$ product, so we report three times the $\times W_O$ product. The counts of the last row follow Section 3.1: each complex-packed input ciphertext costs $2(b-1)$ baby-step rotations plus one conjugation, and each output block of d_1 columns costs g giant-step rotations, with $(n_{\mathrm{in}}, n_{\mathrm{out}}) = (2, 18)$, $(2, 6)$, $(2, 24)$, and $(6, 6)$ for the four products.

Variant	$\times W_{Q/K/V}$	$\times W_O$	$\times W_{FFN1}$	$\times W_{FFN2}$
JKLS, real packing	3744 / 2238*	1248 / 746	4992 / 2960	4974 / 2951
+ Complex packing	2520 / 1505*	840 / 502	3360 / 1986	2496 / 1481
+ SPCMM (Prop. 3.1)	2358 / 913	426 / 241	3144 / 1208	642 / 567
+ Sparse encoding (Prop. 3.2)	2358 / 702	426 / 172	3144 / 928	642 / 359
+ Hoisting output tiles, scale folding	198 / 280	110 / 109	222 / 355	186 / 262
+ Kernel batching and fusion	198 / 82	110 / 47	222 / 122	186 / 95

B Comparison with Prior Work

Setup. Table 10 breaks down the BERT-Base latency by operation, summed over the 12 layers, next to the breakdowns that THOR, Powerformer, MOAI, and Tricycle report. We use SST-2 because it is the only dataset that all four report; the ELLMo column is the SST-2 run of Table 3. The Input

Table 9: Ablation of the CCMM optimizations for one BERT-Base layer, with 128×256 tiles on an A100 80GB. Each row adds the optimizations it names to the row above. Each cell gives the number of key switches / the latency in ms, where each product is timed in isolation on operands at level $\ell = 5$, as the minimum of 10 runs after 3 warm-up runs; in the model the CCMMs run at level 5 to 7 and include the score scaling and masks, so their in-model times in Table 2 are higher. The baby-step size b is 16.

Variant	$Q \times K^\top$	$\times V$
JKLS	1698 / 521	951 / 309
+ Imaginary & Head fold	867 / 283	355 / 126
+ Double hoisting, scale folding	606 / 184	349 / 115
+ Kernel batching and fusion	606 / 99	349 / 79

column gives the shape of each operation: the three input projections run as one fused PCMM, the two attention products run once per head, and the FFN projections expand the hidden dimension from 768 to 3072 and back. Other covers the residual additions, complex packing and unpacking, and weight streaming, which the prior works fold into their operation rows.

Matrix multiplications. The six matrix multiplications take 8.6 s in ELLMo, 17% of the total, against 205.5 s for THOR, 230.9 s for Powerformer, and 60.5 s for Tricycle. These are the per-layer gains of Table 2 carried through the model. The four SPCMMs take 3.6 s and the two CCMMs 5.0 s. The CCMMs cost more per operation because both operands are encrypted, so their transforms cannot be computed offline, and because they run at a higher level than the PCMMs. MOAI is faster on $\times V$ and $\times W_O$, but it amortizes its latency over a batch of 256 inputs.

Nonlinear layers. The Softmax, LayerNorm, and GELU of ELLMo take 9.0 s, against 56.2 s for THOR, which also keeps the original nonlinear functions. Softmax is the largest at 4.93 s, because ELLMo evaluates the original exponential and reciprocal with the iterative method of Appendix A.2.2. Powerformer replaces the Softmax with a fine-tuned surrogate that takes 0.75 s; such surrogates are orthogonal to ELLMo and would save about 4.2 s, at the cost of fine-tuning and accuracy. The two LayerNorms take 1.6 s and GELU 2.4 s, against 9.9 s and 7.3 s in Tricycle. Bootstrapping takes 30.1 s, 60% of the total, against 337.9 s for THOR, 103.7 s for Powerformer, and 227.8 s for MOAI, so it is the main cost of ELLMo. Tricycle spends less on bootstrapping, 18.1 s, but more on the matrix multiplications, and it is 2.0× slower overall.

Other frameworks. SPCMM and the calibrator are compatible with compiler frameworks such as CERIUM (Jayashankar et al., 2025), which reports no breakdown. Terrazzo (Shokri & Tsoutsos, 2026) is not directly comparable, because it uses a different FHE scheme, executes batched inference, and does not report accuracy on a dataset.

Table 10: BERT-Base SST-2 inference time in seconds for one 128-token input on a single A100 80GB, summed over the 12 layers. THOR, Powerformer, MOAI, and Tricycle numbers are as reported by the respective papers; MOAI batches 256 inputs and reports the amortized latency. Other covers the residual additions, complex packing and unpacking, and weight streaming. ELLMo is 12.0× faster than THOR, 6.9× faster than Powerformer, 5.7× faster than MOAI, and 2.0× faster than Tricycle.

Operation	**Input**	**THOR**	**Powerformer**	**MOAI**	**Tricycle**	**ELLMo**
$\times W_{Q/K/V}$	$3 \times (\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768})$	49.77	57.65	16.54	20.04	0.96
$Q \times K^\top$	$12 \times (\mathbb{R}^{128\times64} \times \mathbb{R}^{64\times128})$	16.25	28.76	14.07	3.95	2.47
Softmax	$12 \times (\mathbb{R}^{128\times128})$	15.53	0.75	2.19	3.96	4.93
$\times V$	$12 \times (\mathbb{R}^{128\times128} \times \mathbb{R}^{128\times64})$	13.08	18.95	0.29	1.68	2.51
$\times W_O$	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times768}$	27.43	22.54	0.48	4.96	0.54
LN1	$\mathbb{R}^{128\times768}$	7.13	0.37	0.15	4.93	0.75
$\times W_{FFN1}$	$\mathbb{R}^{128\times768} \times \mathbb{R}^{768\times3072}$	49.80	59.21	16.18	17.52	1.18
GELU	$\mathbb{R}^{128\times3072}$	29.42	8.31	3.3	7.26	2.43
$\times W_{FFN2}$	$\mathbb{R}^{128\times3072} \times \mathbb{R}^{3072\times768}$	49.19	43.77	2.88	12.33	0.93
LN2	$\mathbb{R}^{128\times768}$	4.10	0.30	0.15	4.94	0.86
Pooler	$\mathbb{R}^{128\times768}$	2.70	0.20	0.7	-	0.57
Bootstrap	-	337.86	103.72	227.84	18.14	30.09
Other	-	-	-	-	-	2.01
Total		602.26	344.52	283.95	100.55	**50.23**