BICAMERAL: MEMORY-EFFICIENT TRANSFORMER VIA LATENT TENSOR REMATERIALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Memory capacity limits the size of language models that can be trained and served on given hardware. Quantization lowers the precision of each weight, but leaves the number of trainable parameters unchanged, and with it the gradients and optimizer states that dominate training memory. We introduce **Latent Tensor Rematerialization** (LTR), which generates each weight matrix on demand from compressed latent codes with small learned generators, and in doing so converts a standard matrix into a recomputable activation. This reduces memory usage by directly storing fewer parameters and optimizer states, as an alternative to lowering precision. Moreover, LTR leverages underutilized compute to reduce the number of parameters that must be stored, and its full-precision parameters remain compatible with quantization. We validate the effectiveness of LTR with **Bicameral**, a LLaMA-style transformer architecture whose weight matrices are generated on demand by small decoders. We train Bicameral models with 2B effective parameters to 10B tokens, and find that Bicameral matches the uncompressed baseline on perplexity and zero-shot benchmarks, while reducing stored parameters by 50% and training memory usage by up to 48%.

1 INTRODUCTION

Language-model quality improves predictably with parameter count (Kaplan et al., 2020; Hoffmann et al., 2022), and memory capacity is one of the main limits on how many parameters can be trained and served on given hardware. In training, the weights themselves are the smaller part of this memory. With AdamW and mixed precision, every trainable parameter carries an FP32 master copy, gradient and two optimizer moments, 16 bytes in total, against the 2 bytes of the BF16 weight used in the forward pass (Micikevicius et al., 2018; Rajbhandari et al., 2020). This means that a 2B-parameter model needs 33 GB of memory to train, excluding activations.

Quantization and quantization-aware training reduce memory usage by reducing the precision of weights used in the forward pass (Frantar et al., 2023; Lin et al., 2024; Liu et al., 2024b; Peng et al., 2023), but the number of trainable parameters remains unchanged, and optimizer states often remain in full precision. This means that while quantization reduces *inference* memory usage, memory footprint during training remains large.

Meanwhile, compute is often underutilized. Decoding at small batch sizes performs about one FLOP per byte of weights read, whereas an H200 needs about 200 FLOPs/byte to be compute-bound (NVIDIA, 2024; Williams et al., 2009; Pope et al., 2023), and even well-optimized training runs reach only 38–55% model FLOP utilization (MFU) (Chowdhery et al., 2023; Grattafiori et al., 2024; Jiang et al., 2024).

Combining both observations on memory and compute, we propose that spare compute could be leveraged to **generate** larger weight matrices from fewer stored parameters, effectively performing model decompression. This reduces the model's memory footprint both during inference by reducing persistent memory footprint, and during training, because only stored parameters maintain optimizer states.

We call this *Latent Tensor Rematerialization* (LTR): each weight matrix is produced on demand by a small learned generator from compact latent codes, and treated as an activation rather than a parameter—computed when needed, discarded, and recomputed for the backward pass, similar

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.

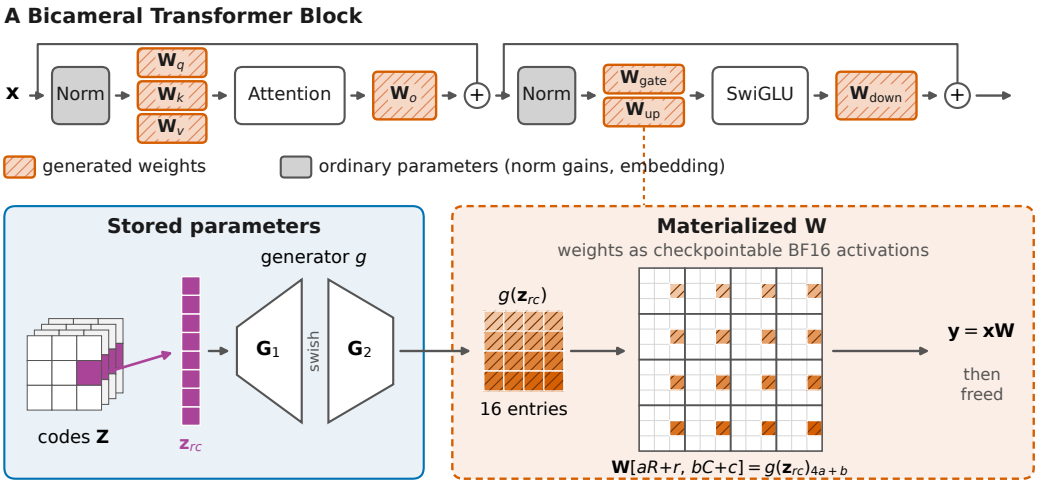


Figure 1: **Top**: all seven projections of every block are generated from latent codes. At $2\times$ and $4\times$ compression each stored code value generates two and four matrix weights respectively. **Bottom**: rematerialization process of one matrix. Only the stored parameters are trainable and maintain optimizer states; each code $\mathbf{z}_{rc}$, one Q-vector of $\mathbf{Z}$, is decoded into 16 entries, placed at (r, c) of each of the 16 large tiles of $\mathbf{W}$, which exists in BF16 and is freed after use.

to activation checkpointing (Chen et al., 2016). Because only the latent codes and generators are parameters, LTR shrinks the master weights, gradients and both optimizer moments together, and the model is stored in its compressed form; since it reduces the number of parameters rather than their precision, it can be combined with quantization to reduce memory usage even further. We validate LTR's feasibility with **Bicameral**, a LLaMA-style transformer architecture in which every projection matrix is generated by a tiny private generator through a *global* tiling (Figure 1).

We make the following contributions:

- **Latent Tensor Rematerialization**, which turns weight matrices into recomputable activations, so that a model's memory usage scales with its compact latent codes rather than with the size of the larger rematerialized model, in both training and inference (Section 3).
- **Bicameral**, which realizes LTR with tiny per-matrix generators of a few thousand parameters each, and a global tiling approach that lets a small generator produce matrices of any size. Bicameral validates LTR by showing that a transformer with materialized weights matches the training performance of its standard baseline while storing **50%** fewer parameters, and outperforms standard transformers with a similar number of stored parameters (Sections 3.2 and 4).
- **Optimized kernels** that accelerate Bicameral by generating and consuming weights on the fly. Inference with our optimized kernel uses up to 70% less memory than the standard model and decodes up to 31% faster at $4\times$ compression (Sections C and 4.4).

2 Related Work

Quantization and distillation. After training, weights can be quantized to 8 bits (Dettmers et al., 2022a; Xiao et al., 2023), 4 bits (Frantar et al., 2023; Lin et al., 2024) or fewer (Egiazarian et al., 2024; Tseng et al., 2024), and quantization-aware or low-precision training uses FP8, FP4 or ternary weights in the forward pass (Liu et al., 2024b; Bondarenko et al., 2025; Micikevicius et al., 2022; Peng et al., 2023; Rouhani et al., 2023; NVIDIA, 2025; Wang et al., 2023; Ma et al., 2024). All keep the number of trainable parameters, each with a master copy, gradient and two moments (Micikevicius et al., 2018; Rajbhandari et al., 2020); optimizer-state compression (Dettmers et al., 2022b; Shazeer & Stern, 2018; Zhang et al., 2024) shrinks the model's memory footprint during training, but does not lead to more efficient inference. Bicameral reduces the stored parameter count directly, and its

stored full-precision parameters can be further quantized. Distillation (Hinton et al., 2015; Sanh et al., 2019; Gu et al., 2024) and pruning (Frantar & Alistarh, 2023; Sun et al., 2024; Xia et al., 2024; Muralidharan et al., 2024) derive a smaller model from a trained larger one, whereas Bicameral is trained from scratch at its final stored size without requiring a teacher model.

Low-rank and structured matrices. Low-rank methods reduce a model's memory footprint in fine-tuning (Hu et al., 2022; Liu et al., 2024a) and pre-training (Zhao et al., 2024; Lialin et al., 2024; Kamalakara et al., 2022). More generally, structured matrices represent a large weight matrix with fewer parameters, as products of small factors (Novikov et al., 2015; Dao et al., 2022) or through a fixed hash into a shared pool (Chen et al., 2015), and cross-layer sharing (Lan et al., 2020; Dehghani et al., 2019; Bae et al., 2025) likewise executes a larger computation than it stores. Bicameral instead learns a nonlinear generator to rematerialize full-width matrices from stored parameters, which are a learned compressed representation of the larger matrices. We compare Bicameral against low-rank methods at equal parameter count in Section 4.3.

Hypernetworks and weight compression. Hypernetworks generate one model's weights using another model (Ha et al., 2017; Platanios et al., 2018; Brock et al., 2018; Knyazev et al., 2021), including in chunks from learned chunk embeddings (von Oswald et al., 2020), similar to Bicameral's weight generation approach. Static hypernetworks already train convolutional networks from scratch with fewer stored parameters than the kernels they generate, at some cost in accuracy (Ha et al., 2017). For transformers, hypernetworks generate task-specific adapters on top of a full set of ordinary weights (Karimi Mahabadi et al., 2021; Charakorn et al., 2025), and learned decoders compress the weights of an already-trained LLM for inference (Ryu et al., 2025; Tian et al., 2026; Lee et al., 2025; Fan et al., 2026). Bicameral instead generates every projection of the model from compressed latent codes, and can be pre-trained this way from scratch, reducing memory usage during training as well as at inference. To our knowledge, no prior work pre-trains a transformer language model whose weight matrices are both compressed and generated.

Rematerialization. Activation checkpointing recomputes discarded activations in the backward pass (Chen et al., 2016; Jain et al., 2020; Kirisame et al., 2021; Korthikanti et al., 2023); LTR extends this technique to weight matrices by generating them from compressed latent representations.

3 METHOD

3.1 LATENT TENSOR REMATERIALIZATION

For a projection $\mathbf{y} = \mathbf{x}\mathbf{W}$ with $\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, LTR computes the weights from a smaller set of stored parameters,

$$\mathbf{W} = \mathcal{A}\big(g_\phi(\mathbf{Z})\big), \tag{1}$$

where $\mathbf{Z}$ are learned latent codes, g_ϕ is a small learned generator applied to each code, and $\mathcal{A}$ is a fixed map that places the generated values in the matrix. Only $\mathbf{Z}$ and ϕ are parameters; $\mathbf{W}$ is an activation of the generator. The backward pass forms $\partial\mathcal{L}/\partial\mathbf{W} = \mathbf{x}^\top \partial\mathcal{L}/\partial\mathbf{y}$ as usual and applies the generator's vector–Jacobian product; when $\mathbf{W}$ is needed again it is regenerated. The 16 bytes of mixed-precision optimizer state are thus paid per stored parameter, $|\mathbf{Z}| + |\phi|$, rather than per matrix entry, and at inference the model is stored as codes and generators, each at any precision.

LTR trades away additional computation for memory efficiency: a generator with hidden width H that emits t^2 values per Q-dimensional code costs $H(1 + 1/\rho)$ multiply-adds per generated entry, where $\rho = t^2/Q$ is the compression ratio, independent of the number of tokens that use $\mathbf{W}$. It can be amortized over the batch in training, and at inference must be absorbed by compute that memory-bound decoding leaves idle (Section 6).

3.2 THE BICAMERAL TRANSFORMER ARCHITECTURE

Bicameral is a pre-norm LLaMA-style transformer (Touvron et al., 2023) with RMSNorm (Zhang & Sennrich, 2019), rotary embeddings (Su et al., 2024), grouped-query attention (Ainslie et al., 2023), SwiGLU (Shazeer, 2020) and a tied embedding. All seven projections of every block ($\mathbf{W}_{q,k,v,o}$, $\mathbf{W}_{\text{gate,up,down}}$) are generated; the embedding and normalization gains are ordinary parameters and

Table 1: Models trained on 10B tokens. Parameter counts exclude the tied embedding. Validation perplexity (PPL) on 39,072 held-out sequences; zero-shot accuracy (%) is length-normalized for ARC, HellaSwag and PIQA (each answer's log-probability divided by its length in bytes) and plain accuracy for WinoGrande, whose scored text is identical across answers, and LAMBADA, which has no answer options.

Model	Stored	Materialized	PPL	ARC-e	ARC-c	HellaS.	PIQA	WinoG.	LAMB.	Avg.
STD 2B	1,982M	–	**14.52**	48.9	25.1	50.2	**72.4**	**54.1**	49.6	50.0
Bicameral 2×	992M	1,982M	14.56	**50.3**	26.1	**50.9**	71.8	53.7	**50.2**	**50.5**
Bicameral 4×	496M	1,982M	15.72	46.5	**26.6**	46.4	70.5	49.7	46.5	47.7
STD 1B	973M	–	15.91	45.5	25.9	45.2	69.8	51.9	46.0	47.4

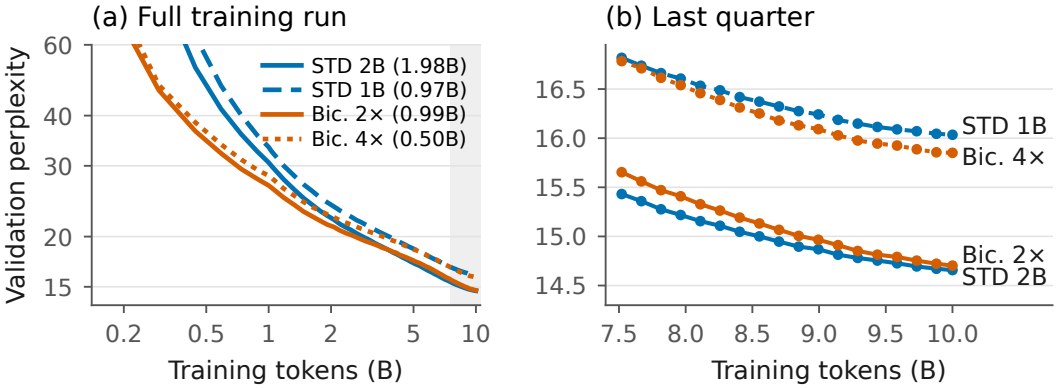


Figure 2: Validation perplexity over 10B training tokens (640-sequence held-out subset). (a) Full run, logarithmic axes. (b) Last quarter of training. Parentheses: stored parameters.

left uncompressed. A Bicameral model executes exactly the computation of the corresponding standard transformer, in which every projection is an ordinary trainable matrix; we call the parameter count of that standard model Bicameral's *effective* size.

Codes and generators. Each projection has its own code grid $\mathbf{Z} \in \mathbb{R}^{R\times C\times Q}$, $R = d_{\text{in}}/t$, $C = d_{\text{out}}/t$, and its own generator

$$g(\mathbf{z}) = \mathbf{G}_2\ \text{swish}_\beta(\mathbf{G}_1\mathbf{z}), \qquad \text{swish}_\beta(u) = u\,\sigma(\beta u), \tag{2}$$

with $\mathbf{G}_1 \in \mathbb{R}^{H\times Q}$, $\mathbf{G}_2 \in \mathbb{R}^{t^2\times H}$ and a learned scalar β. We use $t = 4$, $H = 256$, and $Q = 8$ (2× compressed) or $Q = 4$ (4× compressed). A generator therefore has $256Q + 4097$ parameters; with one generator per matrix and 7 Bicameral matrices per layer, the 140 generators of our 20-layer 2× model take up about 0.087% of its stored parameters.

Global tiling. While previous model compression methods usually group weights into local tiles or vectors (Frantar et al., 2023; Lin et al., 2024; Egiazarian et al., 2024; Tseng et al., 2024), we find that for Bicameral, better performance can be achieved by spreading out generated weights globally. $\mathbf{W}$ is divided into a $t \times t$ grid of large tiles of size $R \times C$, and code $\mathbf{z}_{rc}$ supplies entry (r, c) of every large tile:

$$\mathbf{W}[aR + r,\ bC + c] = g(\mathbf{z}_{rc})_{ta+b}, \qquad a, b \in \{0, \ldots, t-1\}. \tag{3}$$

Each large tile is generated from all RC codes through one generator output (Figure 1). *Local* tiling instead places a code's t^2 values in a contiguous $t \times t$ block. We ablate this design choice in Section 5.2.

4 Experiments

In order to establish their contribution, LTR and Bicameral need to establish the following two claims, which our experiments tested:

- **First**, materialized weights perform as well as the ordinary weights of a model of the same effective size. We show that a Bicameral model that stores half the parameters of a standard transformer reaches the same validation perplexity as that transformer.
- **Second**, training materialized weights is more effective than training a standard transformer with the same number of stored parameters. At an equal token budget, we show that Bicameral outperforms a standard transformer whose parameter count equals Bicameral's stored parameter count.

We establish both in 10B-token pre-training (Section 4.2), show that they carry over to downstream tasks and survive quantization (Section 4.3), and measure the memory Bicameral saves (Section 4.4).

4.1 SETUP

We call a transformer whose projections are ordinary trainable matrices a *standard* transformer (STD). Our main model is a 20-layer STD with $d_{\text{model}} = 3072$, grouped-query attention with 24 query and 6 key–value heads, and a SwiGLU width of 8192; it has 1.98B projection parameters and 2.08B parameters in total (**STD 2B**). Bicameral $2\times$ and $4\times$ execute exactly this computation with $Q = 8$ and $Q = 4$ codes. As the standard transformer with the same stored size as Bicameral $2\times$ we train a 16-layer, width-2048 model (**STD 1B**, 973M vs. 992M non-embedding parameters). All four models see the same 10.0B FineWeb tokens (Penedo et al., 2024) in the same order with the same batch size and schedule; STD models use the conventional AdamW recipe and Bicameral the code learning rate and decay of Section 5.1 (Section A). We report perplexity on 39,072 held-out sequences of 512 tokens; each configuration is trained once, and seed and evaluation noise are small next to the effects we report (Section B).

4.2 MATERIALIZED WEIGHTS VERSUS STANDARD WEIGHTS

Equal effective size. Bicameral $2\times$ reaches a validation perplexity of 14.56 against 14.52 for STD 2B while storing half the parameters (Table 1); halving STD 2B by directly training a smaller architecture instead (STD 1B) raises perplexity to 15.91.

Equal stored size. At the same stored size and token budget, Bicameral $2\times$ outperforms STD 1B, with a stable lead through the second half of training. Bicameral $4\times$, with 51% of STD 1B's stored parameters, also outperforms it. Overall, these results indicate that the extra weights, rematerialized as activations, contribute meaningfully more to training than directly using their stored parameters.

4.3 DOWNSTREAM PERFORMANCE AND QUANTIZATION

Zero-shot accuracy. On ARC (Clark et al., 2018), HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2020) and LAMBADA (Paperno et al., 2016), in the zero-shot formulations of the LM Evaluation Harness (Gao et al., 2024), downstream accuracy follows perplexity (Table 1). Bicameral $2\times$ and STD 2B are indistinguishable on average (50.5 vs. 50.0), but exhibit small differences across individual benchmarks. For example, STD 2B has a small edge on PIQA and WinoGrande, while Bicameral $2\times$ edges out on LAMBADA and HellaSwag. At equal stored size Bicameral $2\times$ outperforms STD 1B on all six benchmarks, by 3.1 points on average, and Bicameral $4\times$ matches STD 1B with 51% of its stored parameters.

Quantization Sensitivity. We apply weight-only round-to-nearest quantization to the projections of the standard models and to Bicameral's stored codes (Figure 3; all conditions in Section D); INT8 is near-lossless for every model. At INT4, Bicameral's latent codes are no more sensitive to quantization than standard weights: perplexity rises by 1.2% ($2\times$) and 2.0% ($4\times$), against 1.5% and 2.5% for STD 2B and STD 1B. However, the generators are the exception: quantizing them as well raises the cost to 9–13%, although they hold only 0.09% of the stored parameters, because each generator's error is applied to every entry of its matrix; on the other hand, keeping generators in FP32 only adds 0.01 bits per weight. As a result, the two forms of compression are compatible and multiplicative (Figure 3b): Bicameral $2\times$ with INT4 codes matches INT4 STD 2B at half its storage (2.1 vs. 4.1 bits per materialized weight), and Bicameral $4\times$ with INT4 codes, at 1.0 bit per weight, is better than INT4 STD 1B at half its storage.

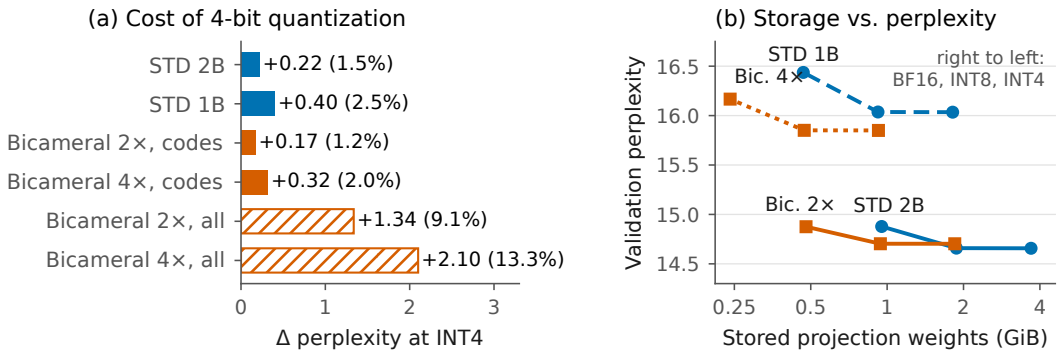


Figure 3: Round-to-nearest weight quantization (groups of 128) of the 10B-token models, measured on a 640-sequence held-out subset. (a) Perplexity increase at INT4; Bicameral quantizes its codes and keeps the generators in FP32, or quantizes both (hatched). INT8 costs at most 0.004 in every case. (b) Perplexity against the storage of the projection weights at different quantization levels.

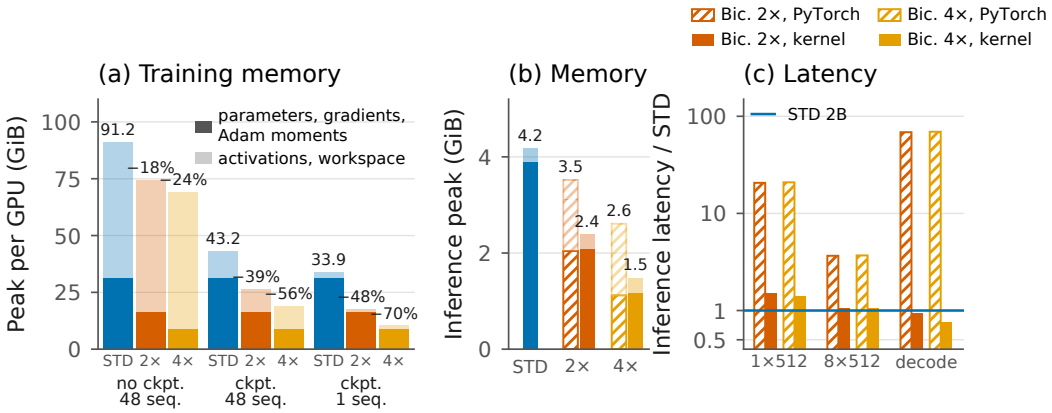


Figure 4: 20-layer models. (a) Peak training memory per H200 with micro-batches of 48 or 1 sequences, without or with per-block checkpointing (ckpt.). (b, c) Inference on one H200 with weights generated by PyTorch (hatched) or our kernels (solid): (b) peak memory of an 8×512 prefill (darker: resident); (c) latency relative to STD 2B.

Generation versus low-precision training. We also compare Bicameral with training the STD weights directly in FP8 or MXFP4, a stronger baseline than post-training quantization, and with a low-rank factorization of the same parameter count, at matched bits per weight (20 layers, 1B tokens; Table 3). Low-precision training is nearly free for the standard model (24.00 to 24.02 perplexity at 8 bits, 24.27 at 4.25), yet Bicameral is better at both budgets (22.34 and 22.66) and low rank is worst (26.11 and 28.96); unlike FP8 or MXFP4, Bicameral also shrinks the master weights, gradients and optimizer states.

4.4 Memory and Optimized Kernels

Training. Two materialization schedules make generation cheap in training: *per-update amortization* generates every $\mathbf{W}$ once per optimizer update, reuses it across the micro-batches of gradient accumulation and applies the generator's vector–Jacobian product once, which gives the same gradients as regenerating per micro-batch; *per-call rematerialization* regenerates each block's weights inside its activation checkpoint, so at most one block's weights are resident. Without checkpointing, the configuration of our 10B-token runs, peak memory falls (Figure 4a) by 18% and 24% at 1.1% and 0.6% more time per update; with per-block checkpointing it falls by 39% and 56% at 13% and 11% more time, and by 48% and 70% with one sequence per micro-batch.

Optimized kernels. Generated with standard PyTorch operations, each generator writes its 256-wide hidden layer to memory for every code. Our inference kernels avoid this (Section C): for prefill, a tensor-core kernel evaluates each generator as a fitted polynomial in its code and writes $\mathbf{W}$ in BF16 for cuBLAS; for decoding, a second kernel multiplies generated values directly into the activation without writing $\mathbf{W}$. Against PyTorch generation in the same compiled model (Figure 4b,c), the kernels make prefill 14–15$\times$ faster for one 512-token sequence and 3.5$\times$ faster for eight and decoding 73–91$\times$ faster. They also reduce memory: the peak of an 8$\times$512 prefill drops from 3.5 to 2.4 GiB at 2$\times$ and from 2.6 to 1.5 GiB at 4$\times$, against 4.2 GiB for STD 2B, and each decoding step needs 1.1–1.2 GiB less. With the kernels, Bicameral 2$\times$ and 4$\times$ hold 2.1 and 1.2 GiB against 3.9 GiB and decode 7% and 31% faster than STD 2B.

5 Ablations

We ablate two design choices of Bicameral: the weight decay applied to the latent codes (Section 5.1), and the arrangement of materialized weights in a weight matrix. Specifically, where the values generated from one latent code are placed in the matrix and how many values each code generates (Section 5.2). All ablations use width-2048 models with 16 or 4 layers at 4$\times$ compression (1.04B effective parameters at 16 layers), trained on 1B or 6B tokens, and each comparison changes only the ablated choice (Section A).

5.1 Weight decay for scale-free codes

AdamW induces weight magnitude growth in Bicameral. Because Bicameral uses materialized weights indirectly optimized by the decoder matrices and the latent code matrices, conventional weight regularizations do not behave as expected. Scaling a matrix's codes by c and its $\mathbf{G}_1$ by $1/c$ leaves $\mathbf{W}$ unchanged, and the value, output, up and down projections feed the residual stream, whose scale the next RMSNorm removes. Bicameral's latent code norm is therefore unconstrained. Additionally, the AdamW optimizer's weight update has a roughly fixed magnitude distribution, and is nearly orthogonal to the parameter, so the norm grows as a random walk (van Laarhoven, 2017; Hoffer et al., 2018; Li & Arora, 2020; Kosson et al., 2024):

$$r_t^2 \;\approx\; r_0^2 + K \sum_{s \le t} \eta_s^2, \tag{4}$$

where r_t is the code RMS, η_s the code learning rate, and $K \approx u^2(1+\beta_1)/(1-\beta_1)$ with $u \approx 0.2$ the RMS of Adam's normalized update per coordinate. With the measured $K = 0.81$, Equation (4) predicts the code RMS of our runs within 7% (Figure 5a). The relative step $u\eta_t/r_t$ then does not depend on the learning-rate multiplier (in a 4-layer Bicameral model, multipliers of 5, 10 and 20 end at the same 3.6–3.7 $\times 10^{-4}$) and decays like $1/\sqrt{t}$ even at constant learning rate: the codes anneal themselves regardless of the schedule. Over 6B tokens their RMS grows 16$\times$, while the weights of the standard model grow only 1.5$\times$ under its weight decay (Figure 5a).

Decay matched to the code learning rate. Decoupled decay $\mathbf{z} \leftarrow (1-\eta_t\lambda)\mathbf{z}$ bounds this growth: balancing $K\eta^2$ against $2\eta\lambda r^2$ gives an equilibrium RMS $r^\star = \sqrt{K\eta/(2\lambda)}$, reached on a time scale $\tau = 1/(2\eta\lambda)$. Codes train at $\alpha\eta$ with $\alpha = 10$, and the usual convention uses decay λ/α to keep the per-step decay fixed; for scale-free codes this leaves $\tau = 50{,}000$ updates at peak learning rate, longer than our runs, and an unreachable equilibrium of 0.60. We instead apply the decay at the codes' own rate, $\mathbf{z} \leftarrow (1-\alpha\eta_t\lambda)\mathbf{z}$ with $\lambda = 1/30$ (decay$_{\text{eq}}$), which shortens τ to 5,000 updates and gives $r^\star = \sqrt{K\alpha\eta/(2\lambda)} = 0.19$. At equilibrium the relative step is $u\alpha\eta_t/r^\star \propto \sqrt{\eta_t}$ and again follows the schedule.

Results. decay$_{\text{eq}}$ improves validation perplexity in all six paired comparisons (Table 5; four of them appear in Figure 6c), by 0.16–0.24, including from 18.60 to 18.42 at 6B tokens. An RMS clamp at $r^\star$ gives a similar gain at 6B tokens (18.44), indicating that bounding the code norm is what matters. The loss improvement arrives late (Figure 5b): decay$_{\text{eq}}$ trails while the learning rate is high, because its codes take relatively larger steps, and overtakes in the last 16% of training as those steps anneal.

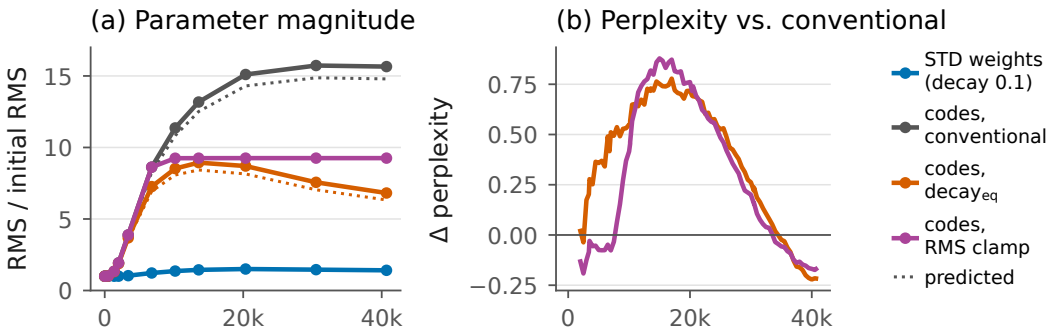


Figure 5: (a) Parameter RMS relative to initialization over 6B tokens: codes of the 16-layer $4\times$ model under conventional decay, equilibrium decay (decay_{eq}), and clamping RMS at $r^\star$ (dotted: projected trajectory from Equation (4)). (b) Perplexity relative to conventional decay.

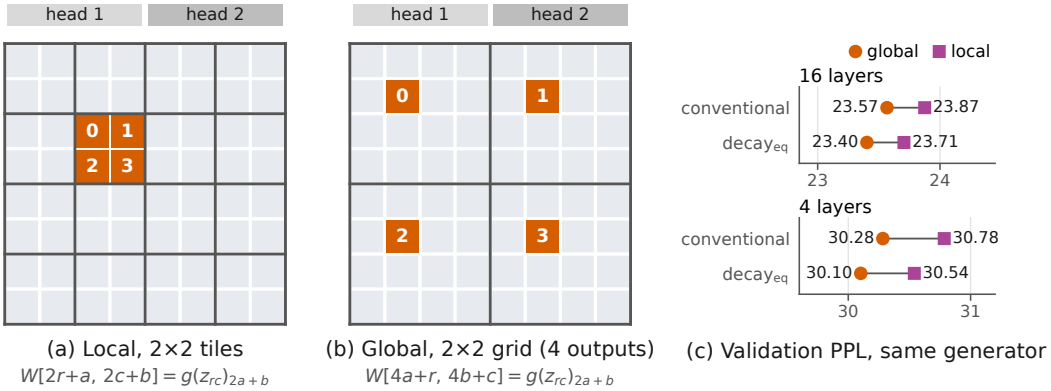


Figure 6: (a, b) Placement of values generated from one latent code in an 8×8 matrix. Local tiling fills one tile; global tiling puts one value in each large tile, so the tile count equals the generator's output width; Bicameral uses a 4×4 grid (16 outputs). (c) Validation perplexity of global and local tiling with the same generator under both code decays ($4\times$ compressed, width 2048, 1B tokens).

5.2 Tiling pattern and tile count

Global versus local tiling. Bicameral generators decode latent codes into tiles, and the arrangement of these tiles can meaningfully impact model performance. A local tile fills a contiguous $t \times t$ block; a global tile acts like a very sparse matrix, and sets the same position in each of t^2 large tiles (Figure 6a,b). The two layouts differ on attention-internal channels, where the local-to-global permutation changes which channels form a head (Section F). In our ablations, we find that global tiling performs better in wider models, which is why it is selected for the deeper and wider 2B models. Specifically, global tiling outperforms local tiling at width 2048, by 0.30–0.50 perplexity (Figure 6c), but underperforms at width 768 with 64-channel heads (Section F).

Tile count. With a matrix's dimensions held constant, we also find the granularity at which the matrix is divided into tiles can affect performance. In global tiling the number of large tiles equals the generator's output width, regardless of matrix dimensions; in local tiling the output width is the tile area, so 16×16 tiles at $4\times$ need a 256-output generator. In our ablations, we find that at fixed $4\times$ compression, 16 global tiles is the optimal granularity as tested (Figure 7a), and local 16×16 tiles are worse than 256 global tiles made by the same generator (Section F).

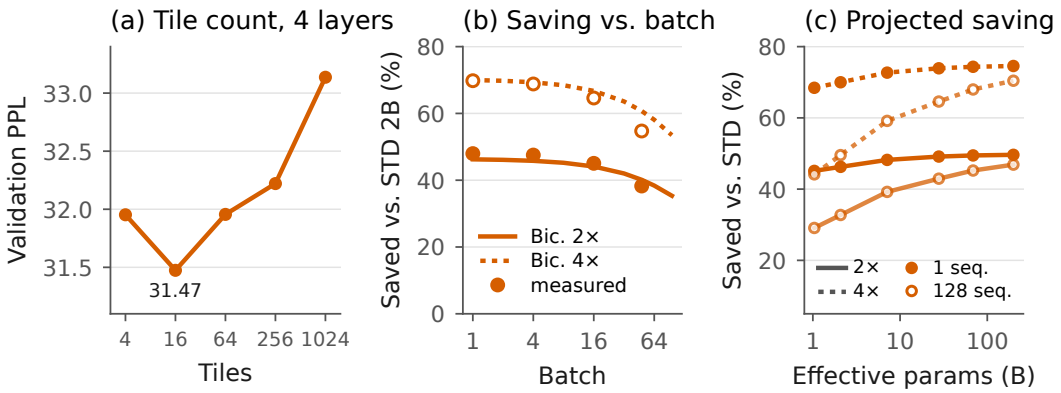


Figure 7: (a) Model performance at different tile counts at 4 layers, $4\times$ compressed, trained to 1B tokens (Table 6). (b) Training memory saved vs. STD 2B with checkpointing: analytic (lines) and profiler (points). (c) Analytic training-memory saving with checkpointing, at micro-batches of 1 and 128 sequences of 512 tokens.

6 Limitations

Computation. Bicameral does not reduce FLOPs: it executes the full computation of its effective-size standard model and adds $H(1 + 1/\rho)$ generator multiply-adds per entry whenever a matrix is materialized, 0.1–2% of the FLOPs of a training update but, at inference, as much as a forward pass over 320–384 tokens. With our fused kernels on one H200 (Figure 4c), a 512-token forward pass is 1.4–1.5$\times$ slower than STD 2B at batch 1 and 4–6% slower at batch 8, while in memory-bound single-token decoding, which reads fewer bytes, Bicameral $2\times$ and $4\times$ decode 7% and 31% faster.

Activation memory. Bicameral does not shrink activations, so its relative memory saving falls as the batch size grows (Figure 7b): with checkpointing, from 48% and 70% at one sequence per micro-batch to 38% and 55% at 48, and to 18–24% without checkpointing in the setting of our 10B-token runs (Figure 4a).

Training horizon and scale. How materialized weights behave beyond 10B tokens and in over-training regimes is undetermined. We have shown via projection that Bicameral's memory saving grows with scale toward 50% and 75%, for $2\times$ and $4\times$ compression respectively (47% and 70% at 200B effective parameters; Figure 7c), but whether its performance holds beyond our 2B models is undetermined.

7 Conclusion

We introduce Latent Tensor Rematerialization, which generates weights as activations from fewer learned parameters, providing a new axis along which to reduce the memory requirement of training and deploying large language models. We validate LTR using the Bicameral transformer architecture, whose performance matches a standard transformer baseline of the same effective size with half the stored parameters, outperforms the baseline with the same stored parameter count, and can work jointly with quantization to further reduce a model's memory footprint.

REPRODUCIBILITY STATEMENT

Section 3.2 and Section A specify the architecture, initialization, optimizer groups and schedules of all runs, and Section B the evaluation protocol and noise levels. We will release the model checkpoints and their training code as well as the fused generator kernels.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods: the model, training and evaluation code, and the fused generator kernels. We used them to design experiments and give feedback on them: agents proposed ablations, controls and configurations, and launched and monitored the training jobs. We used them to propose and refine hypotheses and to interpret results. We used them to develop theoretical models and formulate mathematical claims: the random-walk model of code growth and the weight decay derived from it (Section 5.1), and the relabeling argument that relates global and local tiling, whose proof an agent wrote (Section F). We used them to prepare and audit the training data for corruption and for overlap between the training and held-out splits (Section A).

We did not use generative AI tools to generate synthetic data sets, as this work uses none; translation and qualitative or thematic data analysis are not applicable to this work.

Additionally, we used generative AI tools to write and edit software, to create figures from our results logs, and to organize and summarize related literature and format the references. Agents wrote drafts of the paper from the authors' outline of claims and framing; the authors chose one draft and revised it over several rounds, giving detailed feedback and editing parts directly.

The authors posed the research questions, requested or approved experiments, and decided which results to report and their interpretations; several interpretations proposed by the tools were rejected in this review. Every number in the paper was recomputed from raw experiment outputs (evaluation files and training logs) rather than copied from agent-written summaries, and results produced by agents were audited using their code, configurations and raw outputs. The kernels were validated against reference implementations and gradient checks, and the equivalence between the tiling layouts was verified numerically in FP64. We have reviewed all AI-assisted work, and we take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Sangmin Bae, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Seungyeon Kim, and Tal Schuster. Relaxed recursive transformers: Effective parameter sharing with layer-wise LoRA. In *International Conference on Learning Representations*, 2025.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *AAAI Conference on Artificial Intelligence*, 2020.

Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Low-rank quantization-aware training for LLMs. In *International Conference on Learning Representations (ICLR)*, 2025.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. SMASH: One-shot model architecture search through hypernetworks. In *International Conference on Learning Representations (ICLR)*, 2018.

Rujikorn Charakorn, Edoardo Cetin, Yujin Tang, and Robert Tjarko Lange. Text-to-LoRA: Instant transformer adaption. In *International Conference on Machine Learning (ICML)*, 2025.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, 2015.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Tri Dao, Beidi Chen, Nimit Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, 2022.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *International Conference on Learning Representations (ICLR)*, 2019.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022a.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations (ICLR)*, 2022b.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *International Conference on Machine Learning (ICML)*, 2024.

Feng-Lei Fan, Juntong Fan, Dayang Wang, Jingbo Zhang, Zelin Dong, Shijun Zhang, Ge Wang, and Tieyong Zeng. Hyper-Compression: Model compression via hyperfunction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 48(8):8813–8830, 2026.

Elias Frantar and Dan Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning (ICML)*, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations (ICLR)*, 2023.

Leo Gao, Jonathan Tow, Baber Abbasi, et al. A framework for few-shot language model evaluation, 2024.

Xinyang Geng and Hao Liu. OpenLLaMA: An open reproduction of LLaMA. `https://github.com/openlm-research/open_llama`, 2023.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. In *International Conference on Learning Representations*, 2024.

David Ha, Andrew M Dai, and Quoc V Le. HyperNetworks. In *International Conference on Learning Representations (ICLR)*, 2017.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: Efficient and accurate normalization schemes in deep networks. In *Advances in Neural Information Processing Systems*, 2018.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

Ziheng Jiang, Haibin Lin, Yinmin Zhong, et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *USENIX Symposium on Networked Systems Design and Implementation*, 2024.

Siddhartha Rao Kamalakara, Acyr Locatelli, Bharat Venkitesh, Jimmy Ba, Yarin Gal, and Aidan N. Gomez. Exploring low rank training of deep neural networks. *arXiv preprint arXiv:2209.13569*, 2022.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, and James Henderson. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2021.

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *International Conference on Learning Representations (ICLR)*, 2021.

Boris Knyazev, Michal Drozdzal, Graham W Taylor, and Adriana Romero-Soriano. Parameter prediction for unseen deep architectures. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys*, 2023.

Atli Kosson, Bettina Messmer, and Martin Jaggi. Rotational equilibrium: How weight decay balances learning across neural networks. In *International Conference on Machine Learning*, 2024.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *International Conference on Learning Representations (ICLR)*, 2020.

Byeongheon Lee, Dongkyu Kim, Yeonho You, and Youngmin Kim. LittleBit: Ultra low-bit quantization via latent factorization. In *Advances in Neural Information Processing Systems*, 2025.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *International Conference on Learning Representations*, 2020.

Vladislav Lialin, Sherin Deshpande, and Anna Rumshisky. ReLoRA: High-rank training through low-rank updates. *arXiv preprint arXiv:2307.05695*, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *MLSys*, 2024. Best Paper.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. DoRA: Weight-decomposed low-rank adaptation. In *International Conference on Machine Learning (ICML)*, 2024a.

Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. LLM-QAT: Data-free quantization aware training for large language models. In *Findings of the Association for Computational Linguistics (ACL)*, 2024b.

Shuming Ma, Hongyu Wang, Li Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit LLMs: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations (ICLR)*, 2018.

Paulius Micikevicius, Dusan Stosic, Neil Burgess, et al. FP8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

Saurav Muralidharan, Sharath Turuvekere Sreenivas, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. Compact language models via pruning and knowledge distillation. In *Advances in Neural Information Processing Systems*, 2024.

Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, 2015.

NVIDIA. NVIDIA H200 Tensor Core GPU datasheet. `https://www.nvidia.com/en-us/data-center/h200/`, 2024.

NVIDIA. Pretraining large language models with NVFP4. *arXiv preprint arXiv:2509.25149*, 2025.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Annual Meeting of the Association for Computational Linguistics*, 2016.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024. URL `https://arxiv.org/abs/2406.17557`.

Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang, Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. Fp8-lm: Training fp8 large language models, 2023. URL https://arxiv.org/abs/2310.18313.

Emmanouil Antonios Platanios, Mrinmaya Sachan, Graham Neubig, and Tom Mitchell. Contextual parameter generation for universal neural machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, 2023.

Samyam Rajbhandari, Jeff Rasley, Olatunji Rber, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, et al. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.

Jegwang Ryu, Minkyu Kim, Seungjun Shin, Hee Min Choi, Dokwan Oh, and Jaeho Lee. Neural weight compression for language models. *arXiv preprint arXiv:2510.11234*, 2025.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial Winograd schema challenge at scale. In *AAAI Conference on Artificial Intelligence*, 2020.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Noam Shazeer. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning (ICML)*, 2018.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568, 2024.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Ye Tian, Chengcheng Wang, Jing Han, Yehui Tang, and Kai Han. PocketLLM: Ultimate compression of large language models via meta networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even better LLM quantization with Hadamard incoherence and lattice codebooks. In *International Conference on Machine Learning (ICML)*, 2024.

Twan van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

Johannes von Oswald, Christian Henning, João Sacramento, and Benjamin F Grewe. Continual learning with hypernetworks. In *International Conference on Learning Representations (ICLR)*, 2020.

Hongyu Wang, Shuming Ma, Li Ma, Lei Wang, Wenhui Wang, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. BitNet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. Sheared LLaMA: Accelerating language model pre-training via structured pruning. In *International Conference on Learning Representations*, 2024.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*, 2023.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Annual Meeting of the Association for Computational Linguistics*, 2019.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, 2019.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates, to get better performance. *arXiv preprint arXiv:2406.16793*, 2024.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. In *International Conference on Machine Learning (ICML)*, 2024.

A TRAINING DETAILS

Table 2: Hyperparameters of the 10B-token runs; η is the base learning rate.

Shared	η: peak 3×10^{-4}, 3,391 warmup updates (10%), cosine to 3×10^{-5} at update 33,909. AdamW, $(\beta_1, \beta_2) = (0.9, 0.95)$, $\epsilon = 10^{-8}$, global clipping at 1.0. 576 sequences of 512 tokens per update (2 GPUs $\times$ 6 micro-batches of 48). FP32 parameters, gradients and moments; BF16 matrix multiplications.
STD projections	η, decoupled decay 0.1; init $\mathcal{N}(0, 1/d_{\text{in}})$.
STD embedding, norm gains	η, no decay; embedding init $\mathcal{N}(0, 0.02^2)$.
Bicameral codes $\mathbf{Z}$	10η, $\epsilon = 10^{-9}$, $\mathbf{z} \leftarrow (1 - 10\eta_t/30)\,\mathbf{z}$ per step (decay$_{\text{eq}}$); init $\mathcal{N}(0, 0.0205^2)$.
Generators $\mathbf{G}_1, \mathbf{G}_2$	2η, decoupled decay $1/30$; init $\mathcal{N}(0, 1/Q)$ and $\mathcal{N}(0, 1/256)$.
β, embedding, norm gains	2η, no decay; $\beta = 1$ at init; embedding as STD.

The code decay is implemented as AdamW's own decoupled decay of $\lambda/\alpha = 1/300$ at the code learning rate plus an extra multiplication by $1 - \alpha\eta_t(\lambda - \lambda/\alpha)$ before each step, which together equal decoupled decay $\lambda = 1/30$ at the code learning rate to first order in η; we verified this against a reference AdamW implementation. Bicameral's initialization gives every materialized matrix an RMS of about 0.009, independent of its fan-in, compared with $1/\sqrt{d_{\text{in}}} = 0.018$ for STD 2B; in smaller models, initializing the generated matrices at the standard fan-in scale was slightly worse.

Ablation models. The ablation models in Section 5 have 16 or 4 layers, $d_{\text{model}} = 2048$, 16 query and 4 key–value heads of dimension 128, and a SwiGLU width of 8192. They are trained with the same optimizer and schedule shape on 1B tokens (13,566 updates of 144 sequences, 1,357 warmup updates) or 6B tokens (40,691 updates of 288 sequences, 4,069 warmup updates). The "non-code LR" in Table 5 is the learning-rate multiplier of the generators, embedding and normalization gains.

Matched-bits comparison. The runs of Table 3 use the 20-layer shape, 3,392 updates of 576 sequences (1.0B tokens), 339 warmup updates, and otherwise the recipe of Table 2, without decay$_{\text{eq}}$. FP8 uses E4M3 with one FP32 scale per tensor; MXFP4 uses E2M1 elements with a shared power-of-two scale per 32 values; both keep FP32 master weights and pass gradients with a straight-through estimator. Low-rank ranks are 769 ($\mathbf{W}_{q,o}$), 307 ($\mathbf{W}_{k,v}$) and 1,118 (feed-forward) at the $2\times$ budget, and half of these at $4\times$.

Data. We tokenize the 100B-token sample of FineWeb (Penedo et al., 2024) with the OpenLLaMA v1 tokenizer (Geng & Liu, 2023) (32,000 tokens, byte fallback), separate documents with an end-of-sequence token, and hold out 20M tokens for validation. Because FineWeb deduplicates only within each crawl, 4.4% of held-out tokens belong to documents with an exact copy in the training split; this affects all models equally and does not change relative comparisons.

B NOISE AND REPRODUCIBILITY

The seed-to-seed standard deviation of final validation perplexity for 16-layer, width-2048 Bicameral models at 1B tokens is 0.07 (0.3%). Training is not bitwise reproducible across jobs, and runs with identical configurations can differ by up to about 0.14 perplexity at 1B tokens. Three-seed runs at width 768 give seed standard deviations of 0.1–0.4 perplexity at a perplexity near 40. On the 39,072-sequence evaluation, the sampling standard error is 0.03 for an absolute perplexity and 0.002 for the difference between two models evaluated on the same sequences (STD 2B against Bicameral $2\times$), so evaluation noise is negligible next to training noise.

C MATERIALIZATION AND KERNELS

Training. Our training kernels fuse the generator for the global-16 layout, computing each code's hidden layer, Swish and output in registers and writing the assembled BF16 matrix directly in global

order; a matching backward kernel computes the generator's vector–Jacobian product. With per-update amortization, each update generates every matrix once, keeps the BF16 matrices (3.7 GiB at the 2B-effective size) and an FP32 gradient accumulator for them while the micro-batches run, and applies the backward kernel once. The factor gradients are bitwise identical to those obtained by regenerating the weights for every micro-batch. With per-block checkpointing, the matrices of a block are generated inside the checkpointed function and regenerated in its backward pass, so at most one block's weights are resident.

Inference. For inference, each 16-value generator output is approximated by a polynomial in the code, which a hand-written tensor-core kernel evaluates with per-matrix least-squares coefficients from BF16 codes; the few positions with unusually large codes (below 5×10^{-5} of all positions) receive exact values computed when the kernel is prepared. The basis holds all monomials of degree at most 2 (44 for $Q = 8$, 14 for $Q = 4$), plus quartic terms $(\mathbf{u}^\top \mathbf{z})^4$ along dominant hidden directions for the 20 ($Q = 8$) and 40 ($Q = 4$) of 140 matrices that need them. The generated matrices match the exact FP32 generator to a median relative error of 2.9×10^{-3} for both models, below BF16 resolution, and validation perplexity changes by less than 0.01%. For prefill, the kernel writes each block's $\mathbf{W}$ in BF16 and cuBLAS performs the matrix multiplication; a fused variant that consumes generated tiles from shared memory was correct but slower for 16 or more tokens. For single-token decoding, a second kernel multiplies generated values directly into the activation, so $\mathbf{W}$ is never written to memory; over 48 generated tokens its logits agree with the prefill path to a KL divergence of 2–3 $\times 10^{-4}$, with the same top-1 prediction in 95 of 96 steps. In these measurements Bicameral $2\times$ and $4\times$ hold 2.07 and 1.16 GiB resident, against 3.88 GiB for STD 2B. The PyTorch baseline of Figure 4b,c runs the training code's generator (two matrix multiplications and Swish) and assembly for every projection inside the same compiled model, with BF16 codes, `torch.compile` and CUDA graphs, on the same GPU as the other modes; over 32 decoding steps its logits match the uncompiled FP32 forward to a mean KL divergence of at most 3.2×10^{-4}, with the same top-1 prediction at every step.

Memory measurements. The 48-sequence training peaks in Figure 4a are the maximum allocated GPU memory reported by PyTorch over a full update, with FP32 parameters, gradients and Adam moments, BF16 autocast, compiled models, and 48-sequence micro-batches; the checkpointed runs use one H200 and 12 accumulation steps, the others two H200s and 6 accumulation steps. The analytic model of Figure 7b and c counts 16 bytes per stored parameter, the BF16 block inputs kept by checkpointing, the activations of one block, and, for Bicameral, one block's generated matrices and their gradients; it omits workspace and allocator overhead, and at the 2B shape with 48-sequence micro-batches it predicts a 40% saving for Bicameral $2\times$ against 39% measured. The 1-sequence bars of Figure 4a and the measured points of Figure 7b come from a self-contained profiler, included in the supplementary material, that rebuilds all configurations in plain eager PyTorch; at 48-sequence micro-batches its savings (38% and 55%) agree with the training harness (39% and 56%) to within one point.

D Quantization and downstream results

Table 3: Reducing the bits per materialized weight during pre-training (20 layers, width 3072, 1B tokens). Part of Bicameral's lead at this short horizon comes from its $2\times$ learning rate on the embedding and normalization gains; with those at $1\times$, Bicameral $2\times$ reaches 23.15. Stored counts exclude the embedding; low-rank ranks are chosen to match Bicameral's count.

Budget	Method	Bits/weight	Stored	Val. PPL	Δ PPL
16	STD, BF16	16.00	1,982M	24.00	–
≈8	STD, FP8 training	8.00	1,982M	24.02	+0.03
	Low-rank	8.01	992M	26.11	+2.11
	Bicameral $2\times$	8.01	992M	**22.34**	**−1.66**
≈4	STD, MXFP4 training	4.25	1,982M	24.27	+0.27
	Low-rank	4.00	496M	28.96	+4.96
	Bicameral $4\times$	4.01	496M	**22.66**	**−1.34**

Table 4: All quantization conditions of the 10B-token models: bits and storage of the projection weights (BF16 when unquantized), validation perplexity on the 640-sequence subset, and zero-shot accuracy (%). "Codes" quantizes only Bicameral's codes; "all" also quantizes its generators.

Model	Quantized	Bits	GiB	PPL	ARC-e	ARC-c	HS	PIQA	WG	LAMB.	Avg.
STD 2B	–	16.0	3.69	14.66	48.9	25.1	50.2	72.4	54.1	49.6	50.0
	INT8	8.1	1.87	14.66	49.0	25.0	50.2	72.3	53.7	49.6	50.0
	INT4	4.1	0.95	14.88	48.7	26.4	49.6	72.0	53.4	48.2	49.7
STD 1B	–	16.0	1.81	16.03	45.5	25.9	45.2	69.8	51.9	46.0	47.4
	INT8	8.1	0.92	16.04	45.5	26.0	45.2	69.7	52.0	46.0	47.4
	INT4	4.1	0.47	16.44	45.2	25.1	44.9	69.9	52.7	45.5	47.2
Bicameral 2×	–	8.0	1.85	14.70	50.3	26.1	50.9	71.8	53.7	50.2	50.5
	INT8 codes	4.1	0.94	14.70	50.3	25.8	50.9	71.7	53.7	50.3	50.4
	INT4 codes	2.1	0.48	14.88	49.4	26.5	50.6	71.3	54.9	49.3	50.3
	INT8 all	4.1	0.94	14.71	50.3	26.1	50.9	71.7	53.9	50.2	50.5
	INT4 all	2.1	0.48	16.04	47.9	26.5	49.8	70.7	53.8	47.3	49.3
Bicameral 4×	–	4.0	0.92	15.85	46.5	26.6	46.4	70.5	49.7	46.5	47.7
	INT8 codes	2.0	0.47	15.85	46.3	26.6	46.4	70.3	50.2	46.6	47.7
	INT4 codes	1.0	0.24	16.17	46.3	27.0	46.5	70.9	49.4	44.6	47.5
	INT8 all	2.0	0.47	15.85	46.5	26.7	46.4	70.3	49.7	46.4	47.7
	INT4 all	1.0	0.24	17.95	44.1	26.0	44.8	68.8	50.9	42.2	46.1

E CODE WEIGHT DECAY

Table 5 lists the six paired comparisons of Section 5.1; the four 1B-token comparisons with a 2× non-code learning rate also appear in Figure 6c.

Table 5: Validation perplexity with conventional code decay and with decay_{eq}.

Layers	Tiling	Non-code LR	Tokens	Conventional	decay_{eq}	Δ PPL
16	global	1×	6B	18.60	**18.42**	−0.18
16	global	1×	1B	23.90	**23.73**	−0.17
16	global	2×	1B	23.57	**23.40**	−0.16
16	local	2×	1B	23.87	**23.71**	−0.17
4	global	2×	1B	30.28	**30.10**	−0.18
4	local	2×	1B	30.78	**30.54**	−0.24

F TILING

Kronecker form. With a linear generator $g(\mathbf{z}) = \mathbf{G}\mathbf{z}$, a globally tiled matrix is $\mathbf{W} = \sum_{q=1}^{Q} \mathbf{G}_q \otimes \mathbf{Z}_q$, where $\mathbf{G}_q \in \mathbb{R}^{t\times t}$ is the reshaped q-th column of $\mathbf{G}$ and $\mathbf{Z}_q \in \mathbb{R}^{R\times C}$ the q-th code plane: every large tile is a combination of the same Q full-size code planes. Local tiling reverses the factors, $\mathbf{W} = \sum_q \mathbf{Z}_q \otimes \mathbf{G}_q$. Trained generators are close to linear in their operating range, but not exactly.

Tile count. Table 6 lists the values of Figure 7a, together with a second sweep in which $\mathbf{G}_2$ is rescaled at initialization; the global-versus-local pairs of Figure 6c are the 1B-token rows of Table 5.

Table 6: Tile count at 4 layers, width 2048, $4\times$ compression, 1B tokens (code learning rate $4\times$, other parameters $1\times$): validation perplexity with default initialization, and with $\mathbf{G}_2$ rescaled once at initialization so that the generated weights start at the RMS of the 16-tile model. Parameters are per generator.

Layout	Generator	Parameters	Default init.	Matched init.
Global, 4 tiles	$1 \to 256 \to 4$	1,281	31.95	32.32
Global, 16 tiles	$4 \to 256 \to 16$	5,121	**31.47**	(reference)
Global, 64 tiles	$16 \to 256 \to 64$	20,481	31.96	32.01
Global, 256 tiles	$64 \to 256 \to 256$	81,921	32.22	32.08
Global, 1024 tiles	$256 \to 256 \to 1024$	327,681	33.14	–
Local, 16×16 tiles	$64 \to 256 \to 256$	81,921	32.57	–

Additional results. At 16 layers and width 2048 (1B tokens, code learning rate $4\times$), global-16 reaches 24.22 perplexity against 24.63 for global-64 and 25.47 for local 16×16 tiles. At width 1024 (16 layers, 1B tokens), with $\mathbf{G}_2$ calibrated so that both layouts start at the same weight RMS, global-4 reaches 29.82 and 29.18 perplexity at code learning rates $4\times$ and $8\times$, against 29.91 and 29.54 for global-16. At width 768 (Table 7), local tiling is better at both compression ratios, and Bicameral $2\times$ is slightly better than the standard model with the same computation.

Table 7: Width 768, 12 layers, 500M tokens (33,909 updates), three seeds; validation perplexity on 2,048 held-out sequences.

Model	Stored	Seed 1337	Seed 2027	Seed 4099	Mean
STD	74.3M	38.03	37.71	37.76	37.83
Bicameral $2\times$, global	37.7M	37.74	37.32	37.59	37.55
Bicameral $2\times$, local	37.7M	37.48	37.20	37.55	37.41
Bicameral $4\times$, global	19.0M	41.78	41.56	41.70	41.68
Bicameral $4\times$, local	19.0M	41.13	40.61	41.30	41.01