LATTICELOOP: LEARNING LOOP-SPECIFIC WEIGHTS FROM SHARED LOW-BIT CODES

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped Transformers apply one stack of core layers several times. Full weight tying keeps the model small, but every loop must apply the same weights. Independent weights for each loop remove this restriction, but the core storage grows with the number of loops. Partial untying and per-loop LoRA add less storage, but they limit the change between loops to a few rows or to a low-rank update. Block-scaled low-bit formats such as NVFP4 already split every weight tensor into two parts: 4-bit codes set the pattern of each block of 16 weights, and 8-bit block scales set its magnitude. LatticeLoop shares the codes across loops and learns one table of block scales for each loop. Each additional loop costs 0.5 bits per core weight, one ninth of an independent copy, and every loop runs as one standard block-scaled matrix multiplication. In language model pretraining at matched storage, LatticeLoop closes 92.5% of the loss gap between full tying and independent weights, while partial untying and per-loop LoRA close less than a third of the gap. The advantage holds up to 12 loops and 80B training tokens, and decoding speed stays close to full tying.

1 INTRODUCTION

Looped Transformers apply one stack of core layers several times (Dehghani et al., 2019; Lan et al., 2020; Geiping et al., 2025; Bae et al., 2025b). More loops give more computation without more stored weights. The usual choice is full weight tying: every loop applies the same weights to a changing hidden state. Full tying keeps the model small, but it also limits what the model can learn, and language modeling quality drops compared with a model of the same depth. Independent weights for each loop remove this limit. The cost is storage: the core storage grows with the number of loops, and the main advantage of looping is lost.

Two relaxations sit between these extremes. Partial untying gives a subset of rows or layers its own weights in each loop. Inside that subset the weights can change freely, but all other weights stay identical across loops. Relaxed Recursive Transformers keep the tied weights and add a low-rank LoRA update in each loop (Bae et al., 2025a). The LoRA update can touch every weight, but a small storage budget limits it to a few directions in each matrix. LoRA also adds two small matrix multiplications to every projection at inference. Merging the update into the weights removes this cost, but it stores one full weight copy per loop again. Neither relaxation lets every part of every matrix change at low storage cost.

Low-bit formats offer a different way to spend the extra storage. Block-scaled formats such as NVFP4 store each weight tensor as 4-bit codes, one 8-bit scale for every block of 16 weights, and one tensor scale (Rouhani et al., 2023; Alvarez et al., 2025). The codes set the signs and the relative sizes of the weights inside a block. The block scale sets the magnitude of the whole block. The block scales take only one ninth of the stored bits, but they cover every block of every matrix. Block-scaled 4-bit formats are becoming a standard way to deploy language models on recent GPUs. A looped model deployed in such a format already stores a table of block scales for its core. This raises a simple question: must different loops differ in their codes, or is it enough that they differ in their scales?

We propose LatticeLoop, which answers this question with the second option. LatticeLoop shares one code tensor across all loops and learns one table of block scales for each loop, as Figure 1 shows. Every block gets its own multiplier in every loop. The change between two loops is therefore spread over the whole matrix and is not limited to a few rows or a few directions. Each additional loop costs

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.

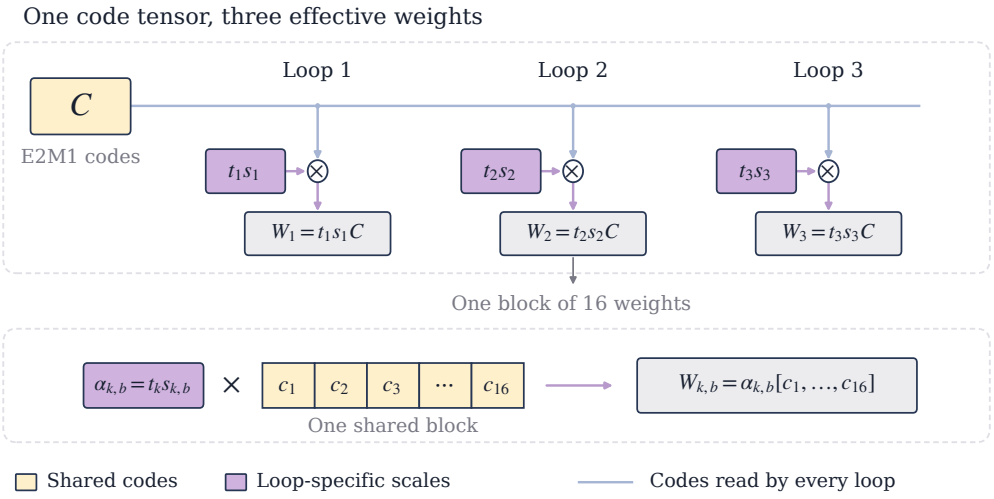

Figure 1: **Shared codes and loop-specific scales.** Top: one E2M1 code tensor C is reused by all three loops. Loop k combines the codes with its E4M3 block scales s_k and FP32 tensor scale t_k. Bottom: the effective scale $\alpha_{k,b} = t_k s_{k,b}$ multiplies the 16 shared codes in block b.

one scale table, which is one ninth of an independent copy. At inference, each loop still runs as one ordinary block-scaled matrix multiplication: the kernel reads the shared codes and the scale table of the current loop. Codes and scales are trained together from random initialization, and all loops start from the same scales, so training starts from the fully tied model.

We pretrain looped Transformers on the Dolma 3 corpus and compare LatticeLoop with full tying, independent weights, partial untying, and per-loop LoRA at matched storage. LatticeLoop recovers most of the quality of independent weights at the storage of the relaxations, and it clearly beats both relaxations in Section 4.2. The advantage holds for deeper loops and longer training in Section 4.3, and LatticeLoop decodes at the speed of full tying in Section 4.4. Section 4.5 explains where the gain comes from: the learned change between loops varies from block to block, and the trained model depends on the loop-specific part of its scales.

2 Related Work

Looped Transformers. Sharing weights across Transformer layers reduces the number of stored weights while keeping the depth of computation, as in ALBERT (Lan et al., 2020) and later studies of cross-layer sharing (Takase & Kiyono, 2023). Universal Transformers apply one transition function repeatedly (Dehghani et al., 2019). Recent looped models use repeated computation for latent reasoning and for scaling test-time compute (Geiping et al., 2025; Bae et al., 2025b). On reasoning tasks, a looped model can match a much deeper model without looping (Saunshi et al., 2025). Weight sharing also helps small models on devices: MobileLLM shares weights between adjacent blocks to add depth without adding storage (Liu et al., 2024b). These models usually tie all weights across loops. Full tying lowers quality compared with a model of the same depth, and independent weights restore quality at a storage cost that grows with the number of loops. LatticeLoop targets the space between these two choices.

Relaxing tied weights. Several methods add loop-dependent parameters to a tied model. Layer-wise or branch-wise scalars on the residual stream stabilize deep looped models (Wang et al., 2026; Li et al., 2026). In parameter-efficient fine-tuning, IA3 rescales activations with learned vectors (Liu et al., 2022), and DoRA splits a weight update into a direction and a magnitude (Liu et al., 2024a). Relaxed Recursive Transformers add a LoRA update (Hu et al., 2022) to each loop of a recursive model (Bae et al., 2025a). VeRA shares one pair of frozen random matrices across layers and learns only small scaling vectors for each layer (Kopiczko et al., 2024). Like LatticeLoop, VeRA keeps a shared tensor and gives each layer cheap scales, but its scales act on a low-rank update instead of on

the stored weights. The LoRA update keeps a shared base and gives each loop its own weights, but an unmerged update adds extra products at inference. Scaling a whole matrix, a row, or a column is another cheap way to make loops differ. LatticeLoop instead gives each block of 16 weights its own scale in each loop, on top of shared codes. Section 4.5 tests how much of the learned change a coarser scaling could represent.

Block-scaled low-bit formats. Microscaling formats store blocks of low-precision values with one shared scale per block (Rouhani et al., 2023; Alvarez et al., 2025). Language models can now be pretrained directly in 4-bit block-scaled formats (NVIDIA, 2025; Castro et al., 2025), so a 4-bit format is a natural way to store the weights of a pretrained model. LatticeLoop builds on this format and does not add a separate one. Recent work quantizes looped models after training. LoopQ optimizes 4-bit weights along the trajectory of the loops (Fang et al., 2026). Ingolfsson et al. (2026) find that per-block activation scales are important for quantized recursive reasoning models. In these methods the scales encode a weight tensor that was trained before. In LatticeLoop, the scales are part of the model: each loop has its own scales, and codes and scales are trained together during pretraining.

3 LATTICELOOP

LatticeLoop shares the weight codes of the looped core across loops and learns one table of block scales for each loop. We first define the weights of each loop, then describe how codes and scales are trained together, how much each sharing rule stores, and how each loop runs at inference.

3.1 WEIGHT REPRESENTATION

Consider a core of L layers that runs for K loops, indexed by $k \in \{1, \dots, K\}$. Each layer has its own projections, and each projection is stored once and applied in all K loops. For a projection with weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, we split each row along the input dimension into blocks of $g = 16$ consecutive weights. Hardware block-scaled matrix multiplication uses the same blocks. The projection has $N = d_{\text{out}} d_{\text{in}}/g$ blocks, indexed by $b \in \{1, \dots, N\}$. In block-scaled formats such as NVFP4 (Rouhani et al., 2023; Alvarez et al., 2025), the weights of block b are

$$W_b = t\, s_b\, C_b, \qquad C_b \in \mathcal{F}_4^g, \quad s_b \in \mathcal{F}_8^+, \quad t > 0. \tag{1}$$

Here $\mathcal{F}_4 = \{0, \pm 0.5, \pm 1, \pm 1.5, \pm 2, \pm 3, \pm 4, \pm 6\}$ is the set of 4-bit E2M1 values, $\mathcal{F}_8^+$ is the set of positive 8-bit E4M3 values with maximum 448, and t is one FP32 scale for the whole tensor.

Full tying shares the codes and both scales across loops. Independent weights store all three for every loop. LatticeLoop shares one code tensor C and gives each loop its own scales:

$$W_{k,b} = \alpha_{k,b}\, C_b, \qquad \alpha_{k,b} = t_k\, s_{k,b} > 0. \tag{2}$$

Here t_k and $s_{k,b}$ are the tensor scale and the block scale of loop k, and $\alpha_{k,b}$ is the effective scale of block b in loop k.

What varies across loops. The codes C_b are the same in all loops, and the effective scale $\alpha_{k,b}$ is positive. Each block therefore keeps its signs, its zero entries, and the ratios between its weights in every loop. Different blocks receive different multipliers in each loop. The difference between two loops is thus spread over the whole matrix and can have full rank. In comparison, a rank-r LoRA update has rank at most r, and partial untying changes only the selected rows.

3.2 JOINT OPTIMIZATION

Codes and scales are learned from random initialization during pretraining. For each projection we keep an FP32 master weight $U \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and a tensor of log gains $A \in \mathbb{R}^{K \times N}$. At every step, a base quantizer described in Appendix B maps U to a tensor scale t_0, block scales $\hat{s}_b \in \mathcal{F}_8^+$, and codes $C_b \in \mathcal{F}_4^g$. The base effective scale of block b is $\beta_b = t_0 \hat{s}_b$. All loops use the codes C. The log gains A set how far the scale of each loop moves away from β.

Loop scales without overflow. The target scale of block b in loop k is

$$\widetilde{\alpha}_{k,b} = \beta_b \exp(A_{k,b}), \tag{3}$$

with $A_{k,b}$ clamped to $[-20, 20]$. To keep every block scale inside the E4M3 range, each loop sets its tensor scale from the largest gain of the projection:

$$t_k = t_0 \exp\Big(\max_{b'} A_{k,b'} \Big), \qquad s_{k,b} = Q_8(\widetilde{\alpha}_{k,b}/t_k)\,, \qquad \alpha_{k,b} = t_k\, s_{k,b}. \tag{4}$$

Q_8 rounds to the nearest value in $\mathcal{F}_8^+$ with ties to even. The argument of Q_8 is at most $\hat{s}_b \le 448$, so no block scale overflows. At initialization $A = 0$, so $\alpha_{k,b} = \beta_b$ in every loop and training starts from the fully tied model.

Gradients through quantization. We pass gradients to U and A with straight-through estimators (Bengio et al., 2013). Let $v_b = \mathrm{clip}(U_b/\beta_b, -6, 6)$ be the normalized weights of block b, and let sg be the stop-gradient operator. We form

$$\overline{C}_b = \mathrm{sg}(C_b) + v_b - \mathrm{sg}(v_b), \qquad \overline{\alpha}_{k,b} = \mathrm{sg}(\alpha_{k,b}) + \widetilde{\alpha}_{k,b} - \mathrm{sg}(\widetilde{\alpha}_{k,b}), \tag{5}$$

and compute each projection with $\overline{W}_{k,b} = \overline{\alpha}_{k,b}\overline{C}_b$. In the forward pass, $\overline{W}_{k,b}$ equals the quantized weight $\alpha_{k,b}C_b$. In the backward pass, $\overline{C}_b$ sends the gradients of all K loops to the shared master weight U, and $\overline{\alpha}_{k,b}$ sends the gradient of loop k to $A_{k,b}$. After training we export only C, the scale tables s_k, and the tensor scales t_k.

Low-precision forward pass. The inputs of all core projections are quantized in blocks of 16 values, and the hidden state is quantized at each boundary between loops. All compared sharing rules use the same quantizers and differ only in how weights are shared across loops.

3.3 Storage and Baseline Budgets

Let P be the number of weights in one copy of the core. The codes take $4P$ bits, and one table of E4M3 block scales takes $0.5P$ bits. Without the small FP32 tensor scales, the core storage is

$$B_{\mathrm{tied}} = 4.5P, \qquad B_{\mathrm{LL}} = (4 + 0.5K)P, \qquad B_{\mathrm{indep}} = 4.5KP. \tag{6}$$

With $K = 3$, the three rules use 4.5, 5.5, and 13.5 bits per core weight. Each additional loop costs LatticeLoop 0.5 bits per core weight, and it costs independent weights 4.5 bits.

We give the two relaxations the same extra budget of $0.5(K-1)P$ bits. Partial untying gives a fraction f of the rows their own weights in loops 2 to K, which costs $4.5(K-1)fP$ bits, so $f = 1/9$. Because all rows are equivalent at random initialization, the untied rows are simply the first rows of each projection. Per-loop LoRA adds BF16 factors to every core projection in loops 2 to K, and the budget allows rank 18. Appendix C lists the exact stored bytes of every model.

Other ways to spend the budget. Three other designs could use the extra storage. Sharing the scales and giving each loop its own codes costs 4 bits per core weight for every added loop, which is eight times the cost of a scale table. A wider tied core with the same storage adds weights to every loop, so it also adds computation to every loop, while LatticeLoop keeps the computation of full tying. Coarser blocks would lower the cost further. MXFP4 uses blocks of 32 values with power-of-two scales (Rouhani et al., 2023), so a loop-specific table would cost 0.25 bits per weight but could only change magnitudes in steps of a factor of two. We use NVFP4 because its blocks of 16 values and its E4M3 scales give the finest magnitude control among the block-scaled formats that current GPUs run natively.

3.4 Execution

Figure 2 shows the model. Each core layer has seven projections: query, key, value, and output in attention, and gate, up, and down in the MLP. All seven projections use shared codes and loop-specific scales. Appendix B describes the quantizers.

An unmerged LoRA projection computes $Wx + L_kR_kx$. It needs two small matrix multiplications in addition to the base projection. Merging L_kR_k into W offline removes these products, but it stores

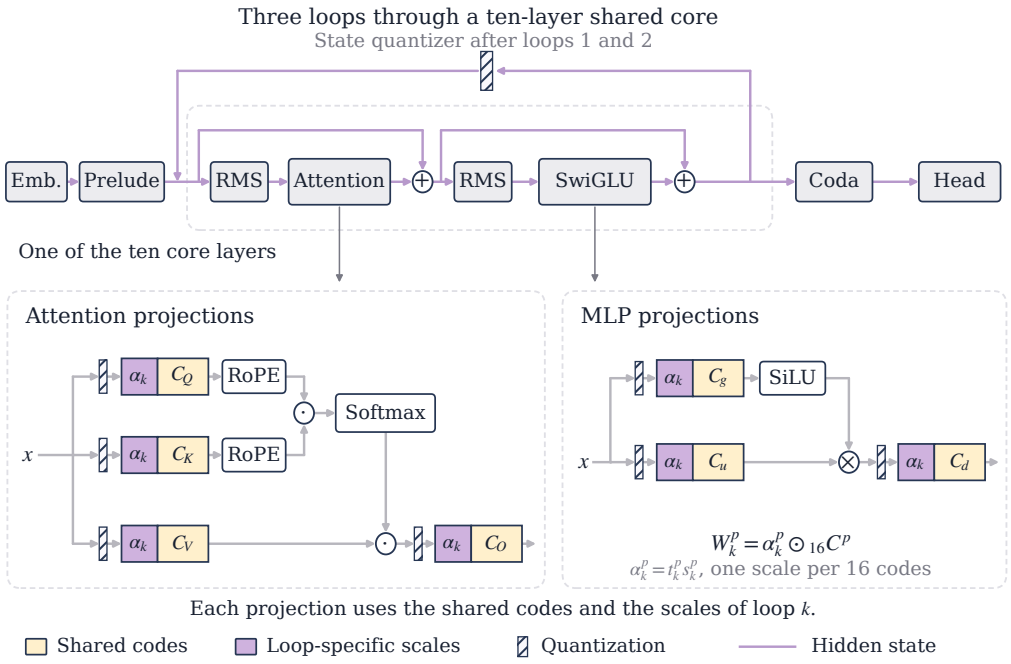

Figure 2: **Looped architecture and scale selection.** Top: one prelude layer, three passes through a ten-layer core, and one coda layer give 32 effective layers. The hidden state is quantized between core passes. Bottom: attention and MLP projections reuse shared codes and select the scales of the current loop.

one full weight copy for every loop. LatticeLoop changes only which scale table the projection reads. A block-scaled matrix multiplication takes the shared codes, the E4M3 scales of loop k, and the tensor scale t_k as inputs. Each loop is therefore one ordinary block-scaled matrix multiplication with no extra branch. Section 4.4 measures the resulting speed.

4 Experiments

The experiments answer four questions. Does LatticeLoop recover the quality of independent weights at the storage of the relaxations? Does the advantage hold for deeper loops and longer training? Does LatticeLoop keep the inference speed of full tying? What do the loop-specific scales learn?

4.1 Setup

Model and training. We train from random initialization with the SmolLM-360M configuration (Hugging Face, 2024): hidden width 960, MLP width 2,560, 15 query heads, and 5 key-value heads of dimension 64. One prelude layer, a ten-layer core run for K loops, and one coda layer give $10K + 2$ layers of effective depth. The core has 70 projections with $P = 98{,}304{,}000$ weights. Embeddings, the output head, the prelude and coda layers, and all normalization weights stay in BF16 in every model. We pretrain on the 150B-token sample of the Dolma 3 Mix (Team Olmo, 2025) with sequence length 2,048, AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay 0.1, and peak learning rate 0.003. The main comparison trains on 20B tokens, and the depth study trains on 20B and 80B tokens. Appendix A gives the full protocol.

Sharing rules. We compare five rules. Full tying shares codes and scales across loops. Independent weights store codes and scales for every loop. LatticeLoop shares codes and learns one scale table per loop. Partial untying and per-loop LoRA use the same extra storage as LatticeLoop, as defined in Section 3.3.

Table 1: **Recovering capacity at a matched storage budget.** All models use $K = 3$ loops, 32 effective layers, and 20B training tokens. Core bits are nominal bits per unique core weight. Payload is the complete weight payload in decimal MB, excluding the manifest. Gap closed follows Equation 7 on held-out log-loss.

	Stored weights		Evaluation		
Method	Core bits	MB	PPL ↓	Gap (%) ↑	Avg. (%) ↑
Full tying	4.5	283.41	15.80	0.0	36.8
Partial untying	5.5	295.69	15.61	30.8	37.9
LoRA ($r = 18$)	5.5	295.62	15.75	7.5	37.1
LatticeLoop	5.5	295.70	**15.23**	**92.5**	**40.5**
Independent weights	13.5	394.00	15.18	100.0	40.9

Bold values identify the best result among the three approximately 5.5-bit methods. Full tying and independent weights are lower- and higher-storage references. Avg. is the unweighted mean over seven zero-shot benchmarks.

Evaluation. We evaluate the exported low-bit weights. Language modeling quality is perplexity on a held-out split of the Dolma 3 Mix that is disjoint from the training data. Downstream quality is the mean zero-shot accuracy on ARC-Easy and ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), OpenBookQA (Mihaylov et al., 2018), PIQA (Bisk et al., 2020), Winogrande (Sakaguchi et al., 2019), and Social-IQA (Sap et al., 2019). To measure how much of the gap between full tying and independent weights a model m recovers, we use the held-out log-loss $\mathcal{L}$:

$$\mathrm{GapClosed}(m) = 100\,\frac{\mathcal{L}_{\mathrm{tied}} - \mathcal{L}_m}{\mathcal{L}_{\mathrm{tied}} - \mathcal{L}_{\mathrm{indep}}}. \tag{7}$$

Full tying and independent weights use the same depth and token budget as model m. Held-out perplexity is our primary measure, and the downstream mean is a secondary check.

4.2 Matched Storage

Table 1 compares three ways to spend one extra bit per core weight at $K = 3$. Full tying and independent weights mark the lower and the upper end of the comparison. LatticeLoop closes 92.5% of the gap between them. Its perplexity and downstream accuracy are close to those of independent weights, while its extra core storage is one ninth of theirs. At the same storage, partial untying and per-loop LoRA close less than a third of the gap. One scale per block and loop is a better use of the extra bit than free rows or a low-rank update.

The three methods differ in where the extra storage can change the weights. Partial untying gives full freedom to one ninth of the rows and no freedom to the rest. Per-loop LoRA reaches every weight, but only along 18 directions in each matrix. LatticeLoop reaches every block of every matrix, with one free magnitude per block. Section 4.5 shows that the change the model learns is indeed spread over individual blocks, which is the kind of change that a row subset or a low-rank update cannot express at this storage.

4.3 Depth and Training Tokens

The advantage of loop-specific scales does not shrink as the model gets deeper or trains longer. Figure 3 and Table 2 increase the number of loops from 3 to 12 at 20B and 80B training tokens. Perplexity and downstream accuracy of LatticeLoop improve with every added loop at both budgets, and the largest gain comes from the first added loops. At every depth and budget where the baselines were trained, LatticeLoop beats full tying with the same depth and per-loop LoRA with the same storage. The storage stays small along the whole curve: with 12 loops, LatticeLoop uses 10 bits per core weight, less than independent weights with only 3 loops. Depth therefore becomes a cheap axis for LatticeLoop: nine additional loops cost as much storage as a single independent copy of the core. Appendix G lists the accuracy on each task.

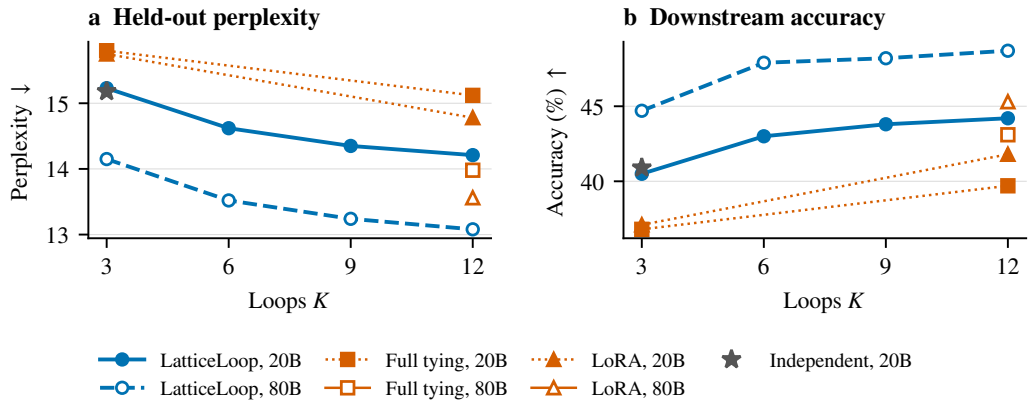

Figure 3: **LatticeLoop improves with every added loop and stays ahead of the baselines at equal depth.** Filled markers: 20B training tokens. Open markers: 80B training tokens. Baselines are shown at the depths where they were trained.

Table 2: **Recurrent depth and training-token scaling.** The same configurations are reported under 20B and 80B token budgets. Effective depth is $10K + 2$. Avg. is the mean zero-shot accuracy in percent. Core bits are nominal, excluding tensor scales and common BF16 weights. Per-task results appear in Appendix G.

	Configuration			20B tokens		80B tokens	
Method	K	Depth	Core bits	PPL ↓	Avg. ↑	PPL ↓	Avg. ↑
Full tying	12	122	4.5	15.12	39.7	13.98	43.1
LoRA ($r = 18$)	12	122	10.0	14.78	41.8	13.56	45.3
LatticeLoop	3	32	5.5	15.23	40.5	14.15	44.7
LatticeLoop	6	62	7.0	14.62	43.0	13.52	47.9
LatticeLoop	9	92	8.5	14.35	43.8	13.24	48.2
LatticeLoop	12	122	10.0	**14.21**	**44.2**	**13.08**	**48.7**

At $K = 12$, LoRA and LatticeLoop have closely matched stored-weight budgets.

4.4 Inference Speed

LatticeLoop is the only relaxation that keeps both the small storage and the speed of full tying. Figure 4 measures decoding throughput on an NVIDIA B200 GPU across batch sizes and depths, and Table 10 in Appendix H lists the absolute prefill and decode numbers. From batch size 1 to 128 and from 3 to 12 loops, LatticeLoop decodes at the speed of full tying. Unmerged LoRA is clearly slower, because every adapted projection adds two small matrix multiplications. Its slowdown grows with depth, since every added loop adds another set of adapter products. Fused multi-adapter kernels such as Punica and S-LoRA reduce the cost of these products (Chen et al., 2024; Sheng et al., 2024), but they still compute products that LatticeLoop does not need. Merging LoRA offline restores the speed, but the stored weights grow to the size of independent weights. Prefill follows the same pattern: LatticeLoop matches full tying, and unmerged LoRA is slower at both depths.

4.5 What the Scales Learn

The previous sections show that loop-specific block scales work. This section asks what the scales learn. The analysis uses a three-loop LatticeLoop model described in Appendix A.

The change between loops is finer than a row or column scaling. If loops differed only by a coarse rescaling, one multiplier per matrix, per row, or per column would describe the change. We fit such multipliers that map the weights of loop 1 to the weights of loops 2 and 3. Table 3 shows

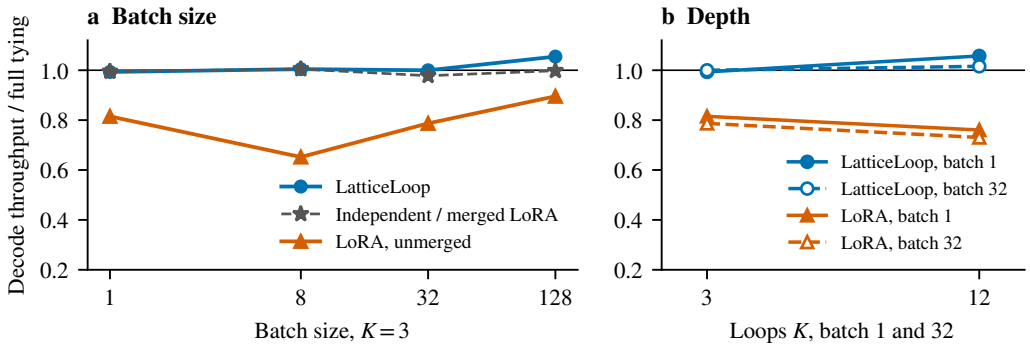

Figure 4: **LatticeLoop decodes at the speed of full tying at every batch size and depth.** Decode throughput relative to full tying on one NVIDIA B200, prompt of 1,024 tokens, 128 generated tokens. Each point is the median of repeated runs.

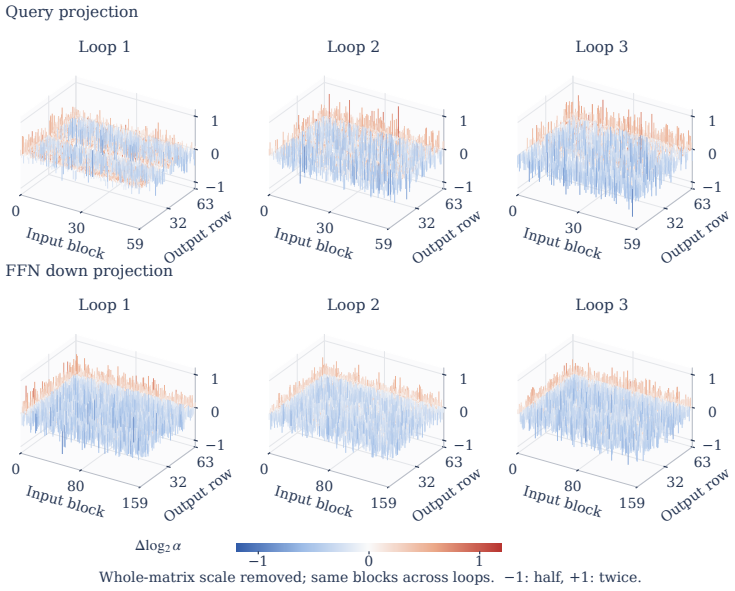

Figure 5: **The scale change between loops varies from block to block.** Log-scale deviation of every block in output rows 0 to 63 of the first core query projection, top, and FFN-down projection, bottom, in the three-loop analysis model. The whole-matrix scale of each loop is removed, and each block is centered over the three loops. A value of $+1$ means twice the block's average scale, and -1 means half.

that the best coarse fit, one multiplier per output row, explains less than a third of the change. The MLP projections carry most of the change, and coarse fits explain even less of it there. Most of the learned change therefore varies from block to block, and a row or column scaling cannot represent it. After removing the scale of the whole matrix, about two thirds of the blocks still change by more than a factor of 1.25 between loops, far more than E4M3 rounding can produce, as Appendix F shows. Figure 5 shows this pattern directly for two projections. The same blocks move up in one loop and down in another, and neighboring blocks in the same row move in different directions.

The model uses the loop-specific scales. We then edit the scale tables of the trained model without further training, as Figure 6 shows. Moving every scale toward the mean over loops raises the held-out loss, and so does moving it away from the mean. Averaging the scales over the three loops removes only the loop-specific part of each scale and keeps the shared part. This edit raises the

Table 3: **How much cross-loop weight change can coarse scaling explain?** Least-squares fits reconstruct loop-2 and loop-3 weights from loop-1 weights over all 70 core projections of the $K = 3$ analysis model. Entries are explained fractions of the squared weight change, in percent.

Multiplier granularity	All	Attention	MLP
Per matrix	20.1	37.3	7.8
Per output row	**29.6**	**46.8**	**17.3**
Per input column	24.0	44.3	9.6
Per 16-element block	100.0	100.0	100.0

Bold marks the best coarse fit. The block-level fit is exact by construction. Attention and MLP projections contribute 41.6% and 58.4% of the total squared weight change, respectively. Under the permutation control of Appendix E, every coarse fit explains at most 0.2%.

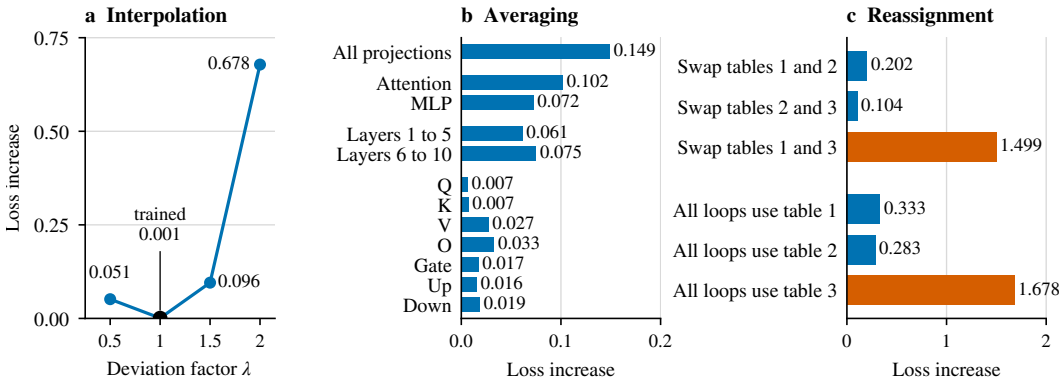

Figure 6: **The trained model relies on the loop-specific part of its scales.** Increase in held-out loss after editing the scale tables of the three-loop analysis model without retraining. a: every log scale moves toward the loop mean for $\lambda < 1$ and away from it for $\lambda > 1$. $\lambda = 1$ is the trained model after re-encoding. b: effective scales are averaged over the three loops in the listed projections. c: complete tables are swapped between loops or copied to every loop.

loss in every group of projections, and over all projections the increase is several times the loss gap between full tying and independent weights at the same training budget. The trained model therefore makes real use of the loop-specific part of its scales. Swapping or copying whole tables raises the loss much more, so the tables are not interchangeable between loops. Figure 6 also shows three finer patterns. The model is more sensitive to enlarging the loop-specific part beyond its trained size than to shrinking it. Averaging hurts more in the attention projections than in the MLP projections, although the MLP projections carry most of the weight change. Swapping the first and the last table costs far more than swapping neighboring tables, so the scales drift gradually from the first loop to the last. Together, the fits, the scale maps, and the edits give one picture: the loops share their codes, and each loop adapts the magnitude of individual blocks in a pattern that the rest of the model depends on.

5 Conclusion

LatticeLoop gives each loop of a looped Transformer its own weights at low storage cost. The loops share the 4-bit codes of the core and differ only in their 8-bit block scales. Each additional loop adds 0.5 bits per core weight and runs as the same block-scaled matrix multiplication as full tying. Training starts from the fully tied model and learns how far the scales of each loop should move away from it. At matched storage, loop-specific block scales recover most of the gap between full tying and independent weights, while partial untying and per-loop LoRA spend the same storage on a few rows or a few directions and recover much less. The advantage holds for deeper models and longer training, and the learned scales change from block to block in a way that the model depends on.

AI USE STATEMENT

Generative AI tools were used solely to assist with writing: polishing the language of author-written text, improving readability, checking the manuscript for consistency and formatting errors, and formatting references. All suggested edits were reviewed and verified by the authors. Generative AI tools were not used for research ideation, formulating hypotheses, designing methods or experiments, implementing code, generating or processing data, analyzing or interpreting results, or creating figures. The authors take full responsibility for the content of this paper.

ETHICS STATEMENT

All models are trained from scratch on the publicly available Dolma corpus and evaluated on standard, publicly available benchmarks. The research does not involve human subjects, crowdsourcing, or the collection of personal data, and we do not release new datasets.

REPRODUCIBILITY STATEMENT

Section 3 defines the weights of every loop, the training of codes and scales, and the storage of every sharing rule. Appendix A lists the architecture, the optimizer, the learning-rate schedule, the token budgets, the training data, and the evaluation protocol. Appendix B specifies the quantizers and the construction of all compared models, and Appendix C gives the exact stored bytes of every model. Appendices D to F define every analysis of Section 4.5, and Appendix H describes the inference implementation and the timing protocol. All training data and downstream benchmarks are publicly available. We will release our code and trained models upon publication.

REFERENCES

Eduardo Alvarez, Omri Almog, Eric Chung, Simon Layton, Dusan Stosic, Ronny Krashinsky, and Kyle Aubrey. Introducing NVFP4 for efficient and accurate low-precision inference. NVIDIA Technical Blog, 2025. URL https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/.

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*, 2025a. URL https://arxiv.org/abs/2410.20672.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron C. Courville, and Se-Young Yun. Mixture-of-Recursions: Learning dynamic recursive depths for adaptive token-level computation. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025b. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b08bbf8b420faa6eeb4020720582ec7-Abstract-Conference.html.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. URL https://arxiv.org/abs/1308.3432.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 7432–7439, 2020. doi: 10.1609/aaai.v34i05.6239. URL https://ojs.aaai.org/index.php/AAAI/article/view/6239.

Roberto L. Castro, Andrei Panferov, Soroush Tabesh, Oliver Sieberling, Jiale Chen, Mahdi Nikdan, Saleh Ashkboos, and Dan Alistarh. Quartet: Native FP4 training can be optimal for large language models. *arXiv preprint arXiv:2505.14669*, 2025.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant LoRA serving. In *Proceedings of Machine Learning and Systems*, 2024.

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, 2018. URL https://arxiv.org/abs/1803.05457.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=B1x1ma4tDr.

Rui Fang, Hsi-Wen Chen, and Ming-Syan Chen. LoopQ: Quantization for recursive transformers, 2026. URL https://arxiv.org/abs/2605.16343.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach, 2025. URL https://arxiv.org/abs/2502.05171.

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*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Hugging Face. SmolLM-360M: Model card, 2024. URL https://huggingface.co/HuggingFaceTB/SmolLM-360M.

Thorir Mar Ingolfsson, Wajeeha Tahir, Anna Tegon, Lionnus Kesting, Gamze İslamoğlu, and Luca Benini. Quantizing recursive reasoning models, 2026. URL https://arxiv.org/abs/2607.16237.

Dawid J. Kopiczko, Tijmen Blankevoort, and Yuki M. Asano. VeRA: Vector-based random matrix adaptation. In *International Conference on Learning Representations*, 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*, 2020. URL https://arxiv.org/abs/1909.11942.

Shuzhen Li, Yifan Zhang, Jiacheng Guo, Quanquan Gu, and Mengdi Wang. DeepLoop: Depth scaling for looped transformers, 2026. URL https://arxiv.org/abs/2607.13491.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A. Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In *Advances in Neural Information Processing Systems*, volume 35. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/0cde695b83bd186c1fd456302888454c-Abstract-Conference.html.

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 *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32100–32121. PMLR, 2024a. URL https://proceedings.mlr.press/v235/liu24bn.html.

Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In *International Conference on Machine Learning*, 2024b.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2381–2391. Association for Computational Linguistics, 2018. doi: 10.18653/v1/D18-1260. URL https://aclanthology.org/D18-1260/.

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

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. In *Advances in Neural Information Processing Systems*, volume 37. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/370df50ccfdf8bde18f8f9c2d9151bda-Abstract-Datasets_and_Benchmarks_Track.html.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, et al. Microscaling data formats for deep learning, 2023. URL https://arxiv.org/abs/2310.10537.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial Winograd schema challenge at scale, 2019. URL https://arxiv.org/abs/1907.10641.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 4463–4473. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1454. URL https://aclanthology.org/D19-1454/.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *International Conference on Learning Representations*, 2025.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-LoRA: Serving thousands of concurrent LoRA adapters. In *Proceedings of Machine Learning and Systems*, 2024.

Sho Takase and Shun Kiyono. Lessons on parameter sharing across layers in transformers. In *Proceedings of the Fourth Workshop on Simple and Efficient Natural Language Processing*, 2023. URL https://arxiv.org/abs/2104.06022.

Team Olmo. Olmo 3. *arXiv preprint arXiv:2512.13961*, 2025.

Shaowen Wang, Bingrui Li, Ge Zhang, Wenhao Huang, Shen Yan, and Jian Li. On the residual scaling of looped transformers: Stability and transferability, 2026. URL https://arxiv.org/abs/2606.18524.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800. Association for Computational Linguistics, 2019. doi: 10.18653/v1/P19-1472. URL https://aclanthology.org/P19-1472/.

A Training and Evaluation Protocol

The main comparison uses $K = 3$ and a 20B training-token budget. The depth study reports $K \in \{3, 6, 9, 12\}$ for LatticeLoop and $K = 12$ controls at 20B and 80B tokens. Table 4 collects the shared architecture, optimizer, and evaluation settings. At a given K, all sharing rules have the same effective depth and differ in their trainable parameters and stored bytes.

Table 4: **Architecture and evaluation settings.** MB denotes 10^6 bytes.

Setting	Value
Initialization	Random, SmolLM-360M configuration
Hidden / MLP width	960 / 2,560
Query / key-value heads	15 / 5, head dimension 64
Prelude / core / coda	1 / 10 / 1 layers, $10K + 2$ effective layers
Core projections	70: Q, K, V, O, gate, up, and down in each core layer
Core weight positions	$P = 98{,}304{,}000$
Common BF16 weights	114,056,640 scalars, 228,113,280 bytes
Training-token budgets	20B; 80B for the extended scaling comparison
Sequence length	2,048 tokens
Optimizer	AdamW, $\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay 0.1
Training corpus	Dolma 3 Mix, 150B-token sample
Held-out data	Dolma 3 Mix held-out split, disjoint from training
Evaluation weights	Exported codes, E4M3 scales, and FP32 tensor scales
Downstream setting	Zero-shot, seven tasks, unweighted mean accuracy

Packing and target accounting. The held-out protocol packs documents with end-of-sequence delimiters and discards an incomplete final sequence. Every retained length-2,048 sequence contributes 2,047 next-token targets. The held-out data is evaluated once, without cycling. The language-modeling loss is averaged over target tokens, and perplexity is its exponential.

Final weights and checkpoints. Evaluation decodes the exported low-bit codes, block scales, and tensor scales. Continuous master weights and log gains are not used in final evaluation.

Analysis model. Section 4.5 and Appendices D to F analyze one $K = 3$ LatticeLoop model with the architecture above. It is trained for 4,600 steps with 1,024 sequences of 2,048 tokens per step, or 9,646,899,200 tokens, on shards 000 to 012 of the FineWeb-Edu 10BT sample (Penedo et al., 2024). Training uses 16 accelerators with 4 sequences per device and 16 gradient-accumulation steps. AdamW uses peak learning rate 0.003, $\beta = (0.9, 0.95)$, weight decay 0.1, gradient clipping at 1.0, 50 warmup steps, and linear decay over the last 920 steps. The block log gains use the same learning rate and no weight decay. The seed is 42. All analyses use its exported weights and shard 013 of the FineWeb-Edu 10BT sample as held-out data: 93,526 packed sequences with 191,447,722 next-token targets.

20B and 80B runs. All runs of Tables 1 and 2 use a global batch of 1,024 sequences of 2,048 tokens, about 2.1M tokens per step. The 20B budget is 9,537 steps and the 80B budget is 38,147 steps. AdamW uses $\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay 0.1, and gradient clipping at 1.0. The learning rate follows a warmup-stable-decay schedule with peak 0.003: linear warmup over the first 1% of steps, a constant phase, and linear decay over the last 10% of steps to one tenth of the peak. The block log gains use the same learning rate and no weight decay. Every model is evaluated at its final step.

Data and downstream evaluation. Training uses the 150B-token sample of the Dolma 3 Mix (Team Olmo, 2025), `allenai/dolma3_mix-150B-1025`, without repetition. Downstream tasks use the lm-evaluation-harness in the zero-shot setting.

B Quantization and Construction of the Compared Models

Base quantizer. For one FP32 master-weight block $U_b \in \mathbb{R}^{16}$, let $m_b = \max_r |U_{b,r}|$. The function Q_4 rounds to the nearest E2M1 value and saturates at magnitude 6. The function Q_8 clips to $[2^{-9}, 448]$ and rounds to the nearest positive E4M3 value. Ties round to even. The reference quantizer computes

$$\begin{aligned} t &= \max\left(\frac{\max_b m_b}{6 \cdot 448}, 2^{-30}\right), \quad & s_b &= Q_8\left(\frac{m_b}{6t}\right), \\ \beta_b &= t s_b, & C_b &= Q_4\left(\frac{U_b}{\beta_b}\right). \end{aligned} \tag{8}$$

The E2M1 magnitudes are $\{0, \frac{1}{2}, 1, \frac{3}{2}, 2, 3, 4, 6\}$, with one sign bit. Lower bounds on t and s_b keep the quantizer defined for zero blocks. These reference quantities are computed without gradients. The straight-through construction of Section 3.2 passes gradients to the continuous weights and the loop-specific gains. The forward effective weights are products of quantized codes and quantized effective scales.

Activations and recurrent state. Inputs to all seven core projections are dynamically quantized in blocks of 16 values. The state quantizer acts after every loop except the last, and fits its block scales at each boundary. Its FP32 tensor scale is fitted once per forward pass from the hidden state at the core input, with a headroom factor of 32. The core-to-coda transition is not quantized. All sharing rules use the same attention, SwiGLU MLP, normalization, and residual connections.

Partial untying and independent weights. Partial untying allocates independent FP32 master weights to the first selected rows in each adapted loop. The selected rows form whole 16-row tiles, with tile counts fixed before training to approximately match LatticeLoop's payload. Untied rows start as copies of the shared rows. The independent-weight model uses the same construction with all rows untied. In both controls, the tensor scale for a projection is fitted to the largest magnitude over its shared and untied weights and is common to the loops. This explains the 280 bytes of tensor scales for these controls in the three-loop exports.

LoRA control. Rank-18 factors are added to every core projection in loops $2, \ldots, K$. The first factor uses Kaiming-uniform initialization and the second starts at zero, so the update is initially zero. Base weights and factors are trained jointly. The export stores the shared base in the tied low-bit format and the factors in BF16, so budget matching counts exported bytes rather than trainable parameters.

C Storage Accounting

Tables 5 and 6 give the exact three-loop exports. Every model additionally contains 228,113,280 bytes of common BF16 weights. The shared codes take $P/2 = 49{,}152{,}000$ bytes, and one E4M3 scale table takes $P/16 = 6{,}144{,}000$ bytes.

Table 5: **Core export components at $K = 3$, in bytes.** The common BF16 payload is identical across methods and is excluded from this component table.

Method	Codes	Block scales	Tensor scales	BF16 adapters
Full tying	49,152,000	6,144,000	280	0
Partial untying	60,067,840	7,508,480	280	0
LoRA	49,152,000	6,144,000	280	12,211,200
LatticeLoop	49,152,000	18,432,000	840	0
Independent weights	147,456,000	18,432,000	280	0

The weight payload of LatticeLoop exceeds full tying by 12,288,560 bytes: two extra block-scale tables plus 560 bytes of tensor scales. Independent weights adds 110,592,000 bytes. Ignoring tensor scales, these increments are one and nine bits per unique core weight position. The payload of LatticeLoop differs from partial untying by 8,240 bytes and from LoRA by 77,360 bytes. Relative to full tying, LatticeLoop's payload grows by 4.34%, while its complete export grows by 4.35%.

Table 6: **Whole-model exports and trainable parameters at $K = 3$.** The complete export is the weight payload plus its JSON manifest. Trainable parameters are counted separately from stored low-bit weights.

Method	Payload bytes	Manifest	Export bytes	Trainable params.
Full tying	283,409,560	80,471	283,490,031	212,360,640
Partial untying	295,689,880	130,473	295,820,353	234,192,320
LoRA	295,620,760	130,484	295,751,244	218,466,240
LatticeLoop	295,698,120	127,290	295,825,410	230,792,640
Independent weights	394,001,560	131,389	394,132,949	408,968,640

D Scale Interventions

Interventions modify the scale tables of the three-loop LatticeLoop analysis model of Appendix A while keeping codes and common BF16 weights fixed. No model is retrained after an intervention. The reference is the unchanged model, and losses are averaged over the full held-out shard of the analysis model. For a projection, $\alpha_{k,b} = t_k s_{k,b}$ denotes the effective scale for block b in loop k.

Re-encoding. Averaging or interpolation produces target effective scales a_b. They are re-encoded as one FP32 tensor scale and E4M3 block scales:

$$\bar{t} = \max\left(\frac{\max_b a_b}{448}, 2^{-30}\right), \qquad \widehat{a}_b = \bar{t}\, Q_8(a_b/\bar{t}). \tag{9}$$

Every modified loop then uses the re-encoded scales $\widehat{a}_b$.

Averaging. For each affected projection, averaging sets $a_b = K^{-1}\sum_k \alpha_{k,b}$ in every loop. The mean is taken over effective scales, including tensor scales, rather than over stored E4M3 entries alone. The reported groups are all 70 core projections, the 40 attention projections, and the 30 MLP projections. At the training budget of the analysis model, full tying and independent weights reach held-out losses of 2.9200 and 2.8936 on the same shard, a gap of 0.026. Averaging over all 70 projections raises the loss of the analysis model by 0.149, more than five times this gap. Figure 6 also averages over core layers 1 to 5, core layers 6 to 10, and the ten instances of each projection type.

Interpolation. Let $c_b = K^{-1}\sum_k \log_2 \alpha_{k,b}$. Interpolation with factor λ sets

$$\log_2 a_{k,b} = c_b + \lambda\big(\log_2 \alpha_{k,b} - c_b\big) \tag{10}$$

in the affected projections and then applies Equation 9. At $\lambda = 1$, any change comes only from re-encoding.

Reassignment. A swap exchanges complete tables, including their block and tensor scales, between two loops in every core projection. With three loops, swapping loops 1 and 3 reverses the scale-table order. Broadcasting gives one loop's complete table to every loop. These operations move stored tables and do not re-encode them.

E Decomposition of the Weight Change

For projection ℓ and loop $k \in \{2, 3\}$, let $W_{\ell,1}$ and $W_{\ell,k}$ be decoded effective weights. A fit partitions the matrices into units u: whole matrices, output rows, input columns, or blocks of 16 weights. The least-squares multiplier for one unit is

$$a_{\ell,k,u} = \frac{\langle W_{\ell,k,u}, W_{\ell,1,u}\rangle}{\max\big(\|W_{\ell,1,u}\|_2^2, 10^{-30}\big)}. \tag{11}$$

A zero unit in loop 1 has a zero numerator and receives multiplier zero. The pooled explained fraction is

$$\mathrm{EF} = 1 - \frac{\sum_{\ell}\sum_{k=2}^{K}\sum_{u}\|W_{\ell,k,u} - a_{\ell,k,u}W_{\ell,1,u}\|_2^2}{\sum_{\ell}\sum_{k=2}^{K}\|W_{\ell,k} - W_{\ell,1}\|_F^2}. \tag{12}$$

Pooling squared errors weights each pair of projection and loop by the magnitude of its weight change. This quantity is a reduction in reconstruction error relative to unchanged loop-1 weights, rather than a centered variance statistic. Because LatticeLoop preserves the code pattern within each block, one block-wise multiplier reconstructs each loop's weights up to floating-point error. The block-level entry of Table 3 is therefore 100% by construction and serves as a reference.

Permutation control. The control replaces $W_{\ell,k}$ by $W_{\ell,1} + \pi(W_{\ell,k} - W_{\ell,1})$, where π shuffles input coordinates within each output row using seed 0. This preserves the entries and squared norm of each row's weight change while removing its coordinate alignment with the loop-1 weights. The same least-squares fits are then applied. Across the nine coarse cells of Table 3, the control explains at most 0.2% of the weight change.

F Normalized Scales and Scale Dispersion

These statistics use all 70 core projections of the three-loop LatticeLoop analysis model of Appendix A. Let $\mathcal{B}_\ell$ be the set of blocks of projection ℓ. We drop the index ℓ from the block scales below.

Removing whole-matrix scale factors. For each loop, subtract the projection's mean log scale:

$$u_{k,b} = \log_2 \alpha_{k,b} - \frac{1}{|\mathcal{B}_\ell|}\sum_{b' \in \mathcal{B}_\ell} \log_2 \alpha_{k,b'}. \tag{13}$$

The normalized ratio of block b is $2^{\max_k u_{k,b} - \min_k u_{k,b}}$. A ratio of one means that the block changes by the same factor as its matrix. The ratio exceeds 1.25 in 4,013,146 of 6,144,000 blocks, or 65.3%.

E4M3 rounding. Adjacent normal E4M3 values differ by at most a factor of 1.125, or 0.17 in $\log_2$. Rounding to the nearest value therefore moves each $\log_2$ scale by at most 0.085, and a uniform rounding error has a root-mean-square value of about 0.049. The spread $\max_k u_{k,b} - \min_k u_{k,b}$ that rounding alone can create is at most 0.17. A normalized ratio above 1.25 requires a spread above 0.32, so rounding cannot produce it. The dispersion values in Table 7 lie between 0.21 and 0.37, four to eight times the rounding error.

Dispersion across loops. To retain whole-matrix factors, center each block over loops and compute

$$\begin{aligned} z_{k,b} &= \log_2 \alpha_{k,b} - \frac{1}{K}\sum_{j=1}^{K} \log_2 \alpha_{j,b}, \\ D_\ell &= \sqrt{\frac{\sum_k \sum_{b \in \mathcal{B}_\ell} z_{k,b}^2}{K|\mathcal{B}_\ell|}}. \end{aligned} \tag{14}$$

A value of zero means identical scales across loops. A value of one means a root-mean-square deviation of one unit in $\log_2$, or a factor of two. Table 7 averages D_ℓ over the ten core layers for each projection type.

Table 7: **Mean scale dispersion across the ten core layers.** Larger values indicate greater scale differences across the three loops.

	Attention				MLP		
	Q	K	V	O	Gate	Up	Down
D_ℓ	0.289	0.300	0.370	0.318	0.208	0.221	0.228

G Full Downstream Results

Tables 8 and 9 list the individual benchmark scores behind Table 2. All values are rounded to one decimal place.

Table 8: **Zero-shot downstream accuracy after 20B training tokens.** Values are percentages. All LoRA models use rank 18. Bold marks the best mean.

Method	K	ARC-E	ARC-C	Hella.	OBQA	PIQA	Wino.	SIQA	Avg.
Full tying	12	44.8	25.1	30.4	29.6	60.8	50.9	36.6	39.7
LoRA	12	48.6	26.9	34.2	31.0	62.9	51.2	38.0	41.8
LatticeLoop	3	45.5	26.4	30.1	31.4	61.3	51.6	37.6	40.5
LatticeLoop	6	51.0	28.0	35.6	32.4	63.8	51.5	38.7	43.0
LatticeLoop	9	53.6	29.2	38.2	30.6	64.0	52.7	38.5	43.8
LatticeLoop	12	52.4	28.5	39.8	32.6	66.5	51.2	38.3	**44.2**

Table 9: **Zero-shot downstream accuracy after 80B training tokens.** Values are percentages. All LoRA models use rank 18. Bold marks the best mean.

Method	K	ARC-E	ARC-C	Hella.	OBQA	PIQA	Wino.	SIQA	Avg.
Full tying	12	49.5	27.2	38.4	31.5	65.2	51.8	37.9	43.1
LoRA	12	53.4	29.5	43.1	33.0	67.5	52.1	38.8	45.3
LatticeLoop	3	52.1	28.7	40.2	32.3	68.4	52.4	39.1	44.7
LatticeLoop	6	58.8	32.1	47.5	33.9	69.8	53.9	39.4	47.9
LatticeLoop	9	58.0	31.4	49.2	35.1	71.2	52.8	39.8	48.2
LatticeLoop	12	57.5	32.3	50.6	35.8	72.1	53.2	39.7	**48.7**

H Execution Details

Training and quality evaluation. Training and quality evaluation simulate NVFP4: weights and activations are quantized and dequantized, and the matrix multiplications run in BF16. All sharing rules use the same path.

Inference implementation. The speed measurements use one NVIDIA B200 with PyTorch 2.10, CUDA 12.8, and FlashInfer 0.6.4. Weight values do not affect speed, so the timed models use random weights with the architecture and storage layout of the trained models. Every core projection runs as a W4A4 NVFP4 matrix multiplication. FlashInfer quantizes the activations online in blocks of 16 values, and `torch._scaled_mm` runs the block-scaled FP4 kernel of cuBLAS for SM100. The FP32 tensor scales multiply the output of the kernel. In LatticeLoop, all loops pass the same code tensor to the kernel, at the same device address, together with the scale table of the current loop. Unmerged LoRA adds two BF16 products and one addition to each projection in loops 2 to K. Offline merged LoRA and independent weights store one NVFP4 copy per loop. The hidden state is quantized between loops, attention uses PyTorch scaled dot-product attention, and the KV cache is BF16. Elementwise operations are fused with `torch.compile`, and the prefill and every decode step are captured as CUDA graphs. The same setup is used for all sharing rules.

Table 10: **Inference speed on one NVIDIA B200.** Prompt of 1,024 tokens and 128 generated tokens. Prefill is at batch size 1. Decode throughput is at batch sizes 1 and 32. Payload is the size of the stored weights.

			Decode (tok./s) ↑	
Method	Payload (MB)	Prefill (ms) ↓	$B = 1$	$B = 32$
$K = 3$, 32 effective layers				
Full tying	283.41	9.36	161.4	3475
LoRA, unmerged	295.62	12.23	131.6	2735
LatticeLoop	295.70	9.37	160.3	3474
LoRA, offline merged	394.00	9.35	160.6	3399
Independent weights	394.00	9.35	160.6	3399
$K = 12$, 122 effective layers				
Full tying	283.41	37.63	39.8	867
LoRA, unmerged	350.57	52.69	30.2	633
LatticeLoop	351.00	36.62	42.1	881

LoRA uses rank 18. Offline merged LoRA and independent weights store one NVFP4 copy per loop and run the same kernels.

Timing protocol. Prefill processes a 1,024-token prompt. Decoding generates 128 tokens greedily, and the decode time includes the argmax of every step. Throughput is the number of generated tokens divided by the decode time. Each measurement uses CUDA events with device synchronization, 10 warm-up runs, and 50 timed runs. For $K = 3$ at batch size 1, the five sharing rules run three times in one process on one GPU, in the orders forward, reverse, and forward, and the table reports the median of all 150 runs. The other settings run once per rule with 50 timed runs. The SM clock stayed at 1,852 MHz in all runs. Replacing `torch._scaled_mm` by the FlashInfer CUTLASS kernel, which applies the tensor scales inside the kernel, changes every $K = 3$ throughput by less than 3%. With this kernel, LatticeLoop stays within 1% of full tying, and unmerged LoRA stays the slowest.

Where decoding time goes. At batch size 1 and $K = 3$, one decode step launches 210 FP4 matrix multiplications in every sharing rule. Activation quantization takes about half of the GPU time and the FP4 matrix multiplications about a quarter. Unmerged LoRA adds 560 kernels for the adapter products, which take 17% of its decode time.

Matrix products per decode step. Each of the ten core layers has seven linear projections. If every projection is counted separately, the core contains $70K$ base projection products per autoregressive step. For rank-r LoRA applied only in loops $2, \ldots, K$, two low-rank products are added per adapted projection:

$$\begin{aligned} N_{\text{base}}(K) &= 70K, \\ N_{\text{LoRA}}(K) &= 70K + 2 \cdot 70(K - 1). \end{aligned} \tag{15}$$

At $K = 3$, this gives 210 base projection products and 490 products for unmerged LoRA. Table 11 counts only the core projections.

Table 11: **Analytical core projection products at $K = 3$.** Counts assume distinct projection products before implementation-level fusion.

Configuration	Base	Adapter	Total
Full tying	210	0	210
LatticeLoop	210	0	210
LoRA, unmerged	210	280	490
LoRA, offline merged	210	0	210
Independent weights	210	0	210