ACTIVATION SPARSITY AS TRAINING–INFERENCE CO-DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Activation sparsity can accelerate transformer inference when a specialized kernel can skip computation associated with zero activations. We ask how broadly and aggressively train-time sparsification should be applied to obtain such exploitable sparsity without excessive quality loss. Using matched Pythia-family pretraining runs, we add fixed-threshold nonlinearities at selected sites in the network forward pass, systematically vary activation pressure, and connect the resulting sparsity to a specialized-kernel execution. We find that extending sparsification across more activation sites can substantially increase model-wide sparsity without reducing latency. At 14M parameters, more than doubling model-wide sparsity from 12.71% to 27.48% slightly increases latency. Conditional kernel analysis identifies the FFN hidden activation h and attention-output projection input z as the most useful sparsification sites. Pressure applied only at h also increases sparsity at z, despite z receiving no direct pressure. Broader pressure produces stronger optimization conflict at less responsive sites. These observations motivate a targeted intervention that achieves lower validation loss than the dense control while reducing latency from 0.652 to 0.573 ms. Scaling analysis confirms the susceptibility to sparsification of h, z but reveals an increasing overhead from the specialized kernel. Our results suggest that sparsification is best treated as a training–inference co-design problem: rather than maximizing sparsity globally, training should induce specific exploitable structure that matches the inference system.

1 INTRODUCTION

Activation sparsity can improve transformer inference efficiency when specialized kernels can skip computation and memory accesses of zero activations. Recently, Cetin et al. (2026) successfully demonstrated inference speedups from train-time pressure and specialized kernel design. This result motivates a follow-up question: *how broadly and aggressively should train-time interventions be applied throughout the transformer to induce exploitable sparsity while reducing latency at an acceptable quality cost?*

Existing approaches differ in where and how they induce sparsity. TEAL applies post-hoc, training-free thresholding to pretrained models; ProSparse and Cetin et al. (2026) apply ℓ_1 regularization to feed-forward (FFN) activations during training; and Q-Sparse applies top-k sparsification to both FFN and attention activations (Liu et al., 2024; Song et al., 2024; Wang et al., 2024). However, these studies do not isolate the effects of different train-time sparsity interventions under a common training protocol. Under matched pretraining conditions using models from the Pythia family (Biderman et al., 2023), we systematically vary the placement and intensity of train-time sparsity interventions across activation sites and measure their effects on model-wide sparsity ($\mathcal{S}_{\text{model}}$) and execution efficiency.

We define an intervention by the placement of activation pressure (P) and forward-pass thresholding (T). To keep the intervention space tractable, for P we introduce Adam-orthogonalized ℓ_1 activation pressure (OL1), limiting conflict with the task update and avoiding an additional pressure-strength hyperparameter. For T, we insert a fixed-threshold nonlinearity at selected activation sites in the model forward pass, with a fixed $\kappa \in \{0, 0.01, 0.05, 0.1, 0.5\}$. The P and T placements are independent, so a recipe such as T_7/P_{all} specifies their respective site sets, while sweeping κ controls sparsification aggressiveness and traces the recipe's trade-off curve. Each model is then paired with a specialized kernel that translates its resulting sparsity into execution benefits.

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.

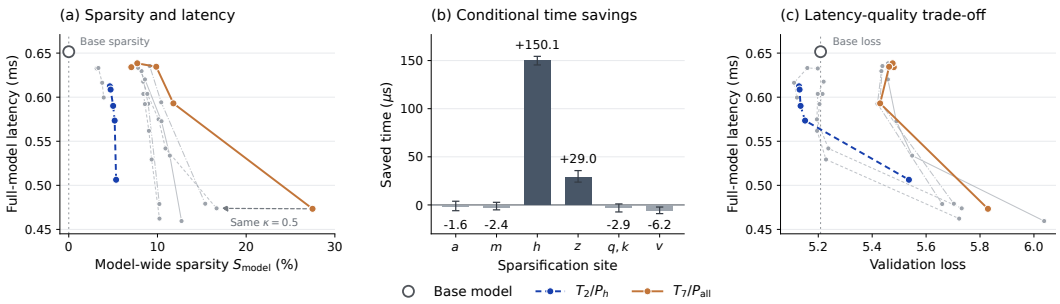

Figure 1: **Sparsity, execution savings and quality on Pythia-14M. (a)** Model-wide logical sparsity $\mathcal{S}_{\text{model}}$ versus full-model latency. The arrow connects T_7/P_{all} to T_7/P_h at $\kappa = 0.5$: 27.48% and 16.66% sparsity yield similar latency. **(b)** Conditional execution ablations on T_7/P_{all} at $\kappa = 0.5$ identify h, z as the profitable skipping sites. Saved time is latency with one skipping path disabled minus latency with all paths enabled. **(c)** Validation loss versus latency exposes the quality cost of broader sparsification. Panels (a,c) show the same 40 checkpoints: T_2/P_h is blue, T_7/P_{all} orange, and other recipes gray. Timings use BF16 on an RTX 5090, batch one and 2,048 tokens.

Across extensive 14M-parameter ablations, we first observe that broader intervention can substantially increase model-wide sparsity without reducing latency. At $\kappa = 0.5$, expanding pressure and thresholding from four to seven sites more than doubles model-wide sparsity, from 12.71% to 27.48%, while latency slightly increases from 0.459 to 0.473 ms (Fig. 1(a)). Decomposing the contribution of individual sites to latency identifies h and z as the sites where sparsity yields meaningful execution benefits (Fig. 1(b)). A per-site sparsity decomposition reveals a related nonlocal effect: applying pressure only at h also increases sparsity at z, despite z receiving no direct pressure (Fig. 3). We call this effect *sparsity spillover*. OL1 provides complementary geometric evidence: broader pressure conflicts more strongly with the task update, especially at q, k, v, suggesting greater resistance and lower susceptibility to sparsification (Fig. 4).

The ablations identify h and z as the most useful sparsification sites. They are more responsive to intervention, and their sparsity directly translates into latency reductions. This motivates a targeted T_2/P_h intervention, which applies pressure only at h and exploits its sparsity spillover to z. At 14M and $\kappa \leq 0.1$, T_2/P_h achieves both lower validation loss than the dense control (baseline) and lower full-model latency (0.573 vs. 0.652 ms, Fig. 1(c)). Finally, we extend this analysis to 31M and 70M models. The same ordering persists: targeted T_2/P_h remains preferable to broader intervention, and localized sparsification continues to benefit execution across scales. However, specialized-kernel overhead grows with model size, making sparsity alone an increasingly insufficient objective.

2 Related Work

2.1 Sparsification interventions

Post-hoc thresholding removes small activations at inference-time of a pre-trained model. While CATS thresholds small-magnitude activations within FFNs (Lee et al., 2024), TEAL extends it to attention (Liu et al., 2024). These studies motivate training-time interventions to adapt activation distributions to sparse computation and improve the trade-off between quality and sparsity.

ReLU Strikes Back studies the effect of activation functions in language models (Mirzadeh et al., 2023), with clear sparsity gains on the use of ReLUs. ProSparse combines progressive ℓ_1 pressure and thresholding at FFNs in continued training (Song et al., 2024). It does not, however, extend its interventions to attention and restricts the analysis to local FFN sparsity. Cetin et al. (2026) extends the ProSparse setup to pre-training and proposes specialized kernels. Both show that training can reshape the quality–sparsity trade-off, but they're restricted to a single local intervention at FFN sites.

Q-Sparse extends sparsification beyond FFNs. It applies top-k selection to attention and FFN projection inputs (Wang et al., 2024), but with no ℓ_1 pressure. However, its top-k operation introduces additional overhead from sorting operations. Spark Transformer address it with statistical top-k (You et al., 2025), but we take a simpler approach and use a constant thresholding paired with pressure.

2.2 Inference speedup

Turning activation sparsity into speedup requires a specialized kernel adapted to the model architecture. TEAL accelerates batch-one autoregressive decoding, reporting up to $1.8\times$ speedup at 50% sparsity (Liu et al., 2024). Cetin et al. (2026) address batched matrix multiplication with their TwELL kernel, and report $1.3\times$ speedup for batch inference. Spark likewise couples its sparsity mechanism with an implementation that reports decoding gains (You et al., 2025). In each setting, skipped computation and memory access must compensate for the additional overhead introduced by the sparse operations. This development has been increasingly accelerated by the use of coding agents, through iterative kernel development with execution and profiling feedback (Ouyang et al., 2025; Luo et al., 2026). We take a related approach and use GPT-6 Astra to develop a specialized kernel based on our training interventions (OpenAI, 2026).

2.3 Positioning Relative to Prior Work

Relative to prior work, our study makes three main contributions. First, rather than fixing a single sparsification mechanism or intervention site, we systematically vary where thresholding and activation pressure are applied under a matched pretraining protocol. Second, we connect these interventions to model-wide rather than purely local sparsity, allowing us to study how pressure at one site can reshape sparsity elsewhere in the network and which sites are more or less susceptible to sparsification. Third, we evaluate the resulting structure through a specialized inference implementation, distinguishing sparsity that actually reduces latency. These contributions motivate a co-design approach in which training induces sparse structures that the inference system can efficiently exploit.

3 Methodology

3.1 Sparsity measure and architectural reach

We define *model-wide sparsity* as $\mathcal{S}_{\text{model}} = N_0/N_{\text{model}}$, where N_0 counts scalar multiplications with a zero activation operand in Pythia-block matrix products, and N_{model} counts all scalar multiplications in those blocks and the final output projection. We aggregate these counts over validation data, excluding masked attention pairs.

To quantify intervention potential a priori, we define the *model-wide sparsity ceiling* as $\mathcal{S}^{\max}_{\text{model}} = N_{\text{reach}}(\mathcal{N})/N_{\text{model}}$, where $N_{\text{reach}}(\mathcal{N})$ counts scalar multiplications that would have a zero activation operand if all selected thresholding sites ($\mathcal{N}$) were 100% sparse. Thus, $\mathcal{S}^{\max}_{\text{model}}$ captures the maximum model-wide sparsity directly reachable through $\mathcal{N}$. It depends on the architecture, selected sites, and sequence length. For a fixed sequence length, the ceiling increases with model size (Table 1). Appendix B gives the precise accounting and definitions.

3.2 Sparsification interventions

A sparsification intervention specifies where activations are thresholded (thresholding sites $\mathcal{N}$) and where activation pressure is applied (pressure sites $\mathcal{P}$). The sets $\mathcal{N}$ and $\mathcal{P}$ need not coincide, and the thresholding and pressure functions may take different forms. Together, these choices define a combinatorial space of sparsification interventions. We restrict this space to keep the computational cost manageable within our compute budget while addressing our central research questions.

For thresholding, we modify the model forward pass by inserting fixed-threshold nonlinearities at the selected $\mathcal{N}$ sites. We avoid the overhead of exact top-k selection (Wang et al., 2024; You et al., 2025) and use one-sided and symmetric operators $G_{+,\kappa}(x) = x\mathbf{1}\{x \geq \kappa\}$ and $G_{\pm,\kappa}(x) = x\mathbf{1}\{|x| \geq \kappa\}$, with $\kappa \geq 0$. The threshold κ is fixed within each training condition and retained at inference. At $\kappa = 0$, $G_{+,\kappa}$ reduces to ReLU, whereas $G_{\pm,\kappa}$ reduces to the identity. We use one-sided thresholding at projection inputs, where signed weight products can recover either sign, and symmetric thresholding for attention operands to better preserve their signed dot-product distribution.

For activation pressure, we define ℓ_1 as the average, across targeted sites and layers, of the mean absolute activation after thresholding. Standard ℓ_1 pressure minimizes $\mathcal{L}_{\text{task}} + \lambda\ell_1$, where λ controls pressure intensity (Song et al., 2024; Cetin et al., 2026). To reduce conflict between the pressure and

task updates and avoid an additional hyperparameter, we introduce orthogonal ℓ_1 pressure (OL1), inspired by gradient-conflict projection methods (Yu et al., 2020; Hsieh et al., 2024). OL1 applies a preconditioned pressure correction after the task-only AdamW update, removes the component opposing the adaptive task direction, and caps its norm relative to the task-update norm.

Intuitively, OL1 removes the ℓ_1 component that opposes the task update and applies a pressure budget b that prevents the sparsification pressure from dominating it. We define r as the pre-cap pressure norm relative to the task-update norm, so r/b measures how much of the available pressure budget the unconstrained correction would use. When $r/b \geq 1$, the cap becomes active, and further increasing λ no longer increases the applied sparsification pressure at a fixed optimizer state. A large opposing component with a high pre-cap norm ratio (r/b) indicates stronger conflict with the task objective. For the multisite study, we fix $\lambda = 1$ and $b = 1$ and measure how strongly different pressure targets conflict with the task update. Appendix A gives the OL1 formulation.

Figure 2 shows the intervention sites within the Pythia architecture. The variables a and m denote the normalized inputs to the attention and FFN branches, respectively; h is the FFN hidden activation; z is the concatenated attention context before the output projection; and q, k, and v are the query, key, and value operands. Attention probabilities are denoted by p. We apply $G_{+,\kappa}$ to selected sites in $\{a, m, h, z\}$ and $G_{\pm,\kappa}$ to selected sites in $\{q, k, v\}$. When h is targeted, the thresholding nonlinearity replaces GELU. Table 1 summarizes the thresholding and pressure assignments, together with their sparsity ceilings across model sizes.

Table 1: **Intervention recipes and analytic sparsity ceilings.** T_1 replaces GeLU with ReLU at h. Pressure uses OL1. For $T_2/T_4/T_7$, we sweep $\kappa \in \{0, .01, .05, .1, .5\}$ with $\lambda = 1$ and $b = 1$. Executed conditions are listed in Appendix C.2.

Recipe	Threshold sites $\mathcal{N}$	Pressure sites $\mathcal{P}$	Reach (%)			
			14M	31M	70M	410M
T_0/P_0 (Base Model)	$\varnothing$	$\varnothing$	0.00	0.00	0.00	0.00
T_1/P_0	h	$\varnothing$	4.28	7.58	12.35	24.93
T_1/P_1	h	h	4.28	7.58	12.35	24.93
T_2/P_h	h, z	h	5.35	9.48	15.44	31.16
T_4/P_0	a, m, h, z	$\varnothing$	12.83	22.75	37.06	74.78
T_4/P_h	a, m, h, z	h	12.83	22.75	37.06	74.78
T_4/P_{all}	a, m, h, z	$\mathcal{N}$	12.83	22.75	37.06	74.78
T_7/P_0	a, m, h, z, q, k, v	$\varnothing$	29.95	37.92	49.42	87.25
T_7/P_h	a, m, h, z, q, k, v	h	29.95	37.92	49.42	87.25
T_7/P_{all}	a, m, h, z, q, k, v	$\mathcal{N}$	29.95	37.92	49.42	87.25

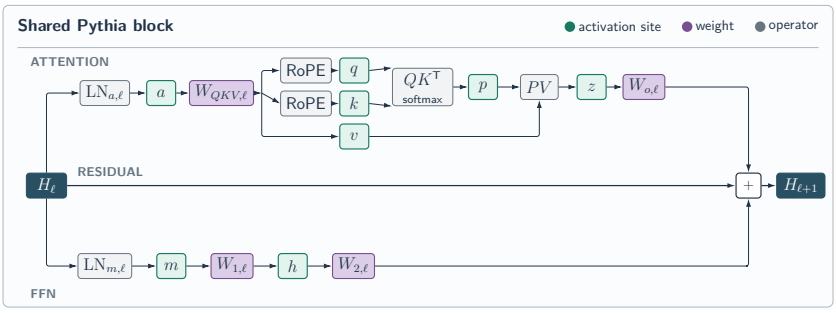

Figure 2: **Intervention sites in a shared Pythia block.** The attention and FFN branches use separate LayerNorms and join the residual path. Green nodes denote activation sites, and purple nodes denote learned weights. Table 1 specifies the thresholding and pressure assignments.

4 EXPERIMENTAL STUDY

We pretrain Pythia-family models at 14M, 31M, 70M, and 410M parameters from random initialization (Biderman et al., 2023) on MiniPile (Kaddour, 2023). Each run receives 712 optimizer steps and 1.493 billion input tokens. Within each model size, runs share initialization, data order, and training budget. We evaluate loss and $\mathcal{S}_{\text{model}}$ on the full validation set. Appendix C provides the full protocol. Our analysis proceeds from extensive 14M ablations, to the non-local sparsity spillover effect, to a site-level analysis linking sparsity, bypassed computation, and latency savings, and finally to a scaling study of targeted and broad thresholding at 14M, 31M, and 70M.

4.1 EXTENSIVE ABLATIONS AT 14M

We compare interventions at 14M, separating aggressive thresholding ($\kappa = 0.5$) from moderate settings ($\kappa = \{0.05, 0.1\}$). First, **broader coverage increases sparsity without consistently reducing latency.** At $\kappa = 0.5$, T_7/P_{all} more than doubles the sparsity of T_4/P_{all} (27.48% versus 12.71%), yet latency slightly increases from 0.459 to 0.473 ms (Table 2). Within T_7, broadening pressure from P_h to P_{all} adds 10.82 percentage points of sparsity with nearly unchanged latency, while loss increases from 5.732 to 5.829. All seven aggressive recipes reduce latency relative to Base, but at the cost of higher loss.

Table 2: **Sparsity and latency under aggressive thresholding.** All seven thresholded 14M recipes at $\kappa = 0.5$. All latencies, including Base, use the specialized 14M kernel on RTX 5090 (BF16, batch one, 2,048 tokens, full logits), with three timing processes per checkpoint.

Recipe	Ceiling (%)	Sparsity (%)	Latency (ms)	Loss
Base (kernel)	0.00	0.00	0.652	5.209
T_2/P_h	5.35	5.34	0.506	5.536
T_4/P_0	12.83	10.22	0.479	5.660
T_4/P_h	12.83	10.23	0.462	5.723
T_4/P_{all}	12.83	12.71	0.459	6.038
T_7/P_0	29.95	15.39	0.479	5.703
T_7/P_h	29.95	16.66	0.474	5.732
T_7/P_{all}	29.95	27.48	0.473	5.829

Moderate thresholding with localized pressure offers a better quality–latency balance. At both moderate thresholds, P_h yields lower loss and latency than P_{all} under both T_4 and T_7 (Table 3). At $\kappa = 0.05$, T_4/P_h reduces latency from Base's 0.652 to 0.562 ms while also reducing loss from 5.209 to 5.196. Narrowing thresholding further to T_2/P_h reduces loss to 5.134.

4.2 SPILLOVER: LOCAL PRESSURE INDUCES NONLOCAL SPARSITY

Pressure at h increases sparsity at z under both T_4 and T_7. At $\kappa = 0.05$, P_h raises h sparsity from 74–85% to 94–99% across layers (Figure 3). Sparsity at the unpressured z site also increases in every layer: from 71–85% to 83–92% under T_4, and from 70–87% to 84–94% under T_7. This spillover is therefore not specific to four-site thresholding and shows that a local intervention at h can induce sparsity at an unpressured downstream site. More broadly, it provides evidence of nonlocal representational adaptation to localized sparsity pressure.

Sparsification susceptibility differs sharply across sites. Under T_4/P_{all}, extending pressure beyond h increases sparsity across targeted sites, with z reaching 99.74–99.85% in layers 2–4. Extending pressure further under T_7/P_{all} yields little additional sparsity at q, k, v (3–13%) while making h and z less sparse than under P_h in every layer. The OL1 geometry in Figure 4 explains this asymmetry: pressure beyond h produces larger pre-cap pressure norms and opposing components to the task update, most strongly for T_7/P_{all}. At $\kappa = 0.05$, the median removed component increases from 0.15–0.17% under P_h to 1.12% under T_4/P_{all} and 66.84% under T_7/P_{all}, while the corresponding pre-cap norm ratios rise from 0.08–0.10 to 0.28 and 46.00. Although this pattern may partly reflect the OL1 mechanism, it is consistent with a, m, q, k, v exhibiting greater conflict with the task objective and greater resistance to sparsification.

Table 3: **Quality–latency trade-offs at moderate thresholds.** All seven thresholded 14M recipes at $\kappa \in \{0.05, 0.1\}$, with the same metrics and timing protocol as Table 2.

Recipe	κ	Ceiling (%)	Sparsity (%)	Latency (ms)	Loss
Base (kernel)	—	0.00	0.00	0.652	5.209
T_2/P_h	0.05	5.35	5.00	0.590	5.134
	0.1	5.35	5.17	0.573	5.151
T_4/P_0	0.05	12.83	8.21	0.630	5.434
	0.1	12.83	8.95	0.593	5.420
T_4/P_h	0.05	12.83	9.04	0.562	5.196
	0.1	12.83	9.34	0.529	5.229
T_4/P_{all}	0.05	12.83	10.46	0.573	5.490
	0.1	12.83	11.39	0.534	5.548
T_7/P_0	0.05	29.95	9.13	0.635	5.438
	0.1	29.95	10.43	0.594	5.429
T_7/P_h	0.05	29.95	10.13	0.575	5.195
	0.1	29.95	10.90	0.542	5.237
T_7/P_{all}	0.05	29.95	9.86	0.635	5.463
	0.1	29.95	11.80	0.593	5.429

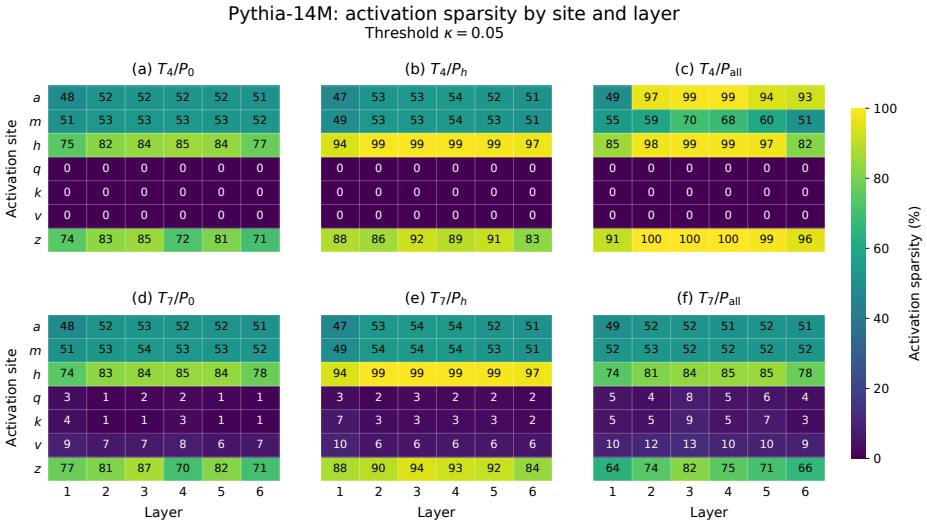

Figure 3: **Activation sparsity by site and layer in Pythia-14M.** The six threshold/pressure recipes at $\kappa = 0.05$: T_4 (top) and T_7 (bottom), each under P_0, P_h and P_{all}. Each cell gives the percentage of zero activations over all complete validation sequences at one site and layer. Values are rounded to whole percentages, so 0 or 100 does not establish a completely dense or all-zero tensor.

4.3 When sparsity translates to speedup

We next test whether the sites most susceptible to sparsification are also those whose sparsity yields the largest execution-time savings. Table 4 isolates each skipping path at 14M by disabling it while holding all others fixed. For T_7/P_{all} at $\kappa = 0.5$, it reports site sparsity, *bypass* (the percentage of omitted matrix instructions), and conditional time saved. We use this recipe because it produces the highest sparsity, maximizing the opportunity to expose execution benefits. The result is consistent with the susceptibility analysis: **latency savings are observed only for sparsity at h and z.**

Sparsity reduces latency only when the avoided computation outweighs the overhead of detecting and skipping it. At h and z, the sparse kernel checks 8×16 activation tiles before loading the associated

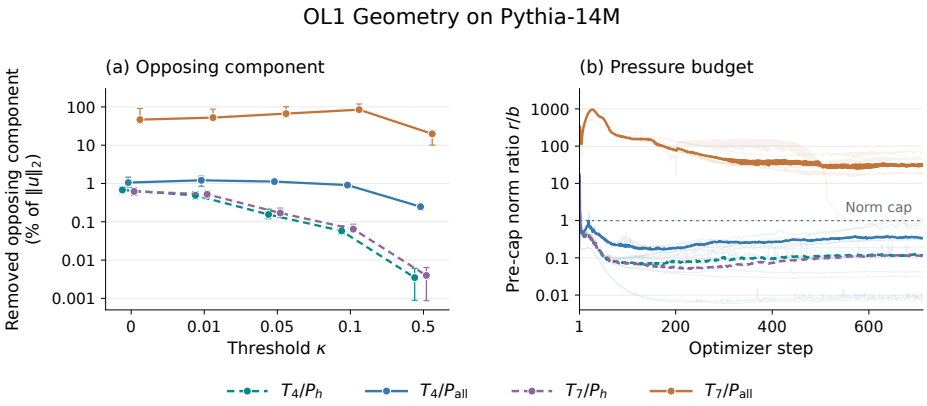

Figure 4: **OL1 geometry reveals greater resistance under broader pressure at 14M. (a)** Median removed opposing component, expressed as a percentage of the task-update norm $\|u\|_2$. **(b)** Pre-cap norm ratio r/b: unconstrained pressure magnitude relative to its available budget. The dotted line at $r/b = 1$ marks the norm cap: values above this threshold exceed the budget and are rescaled before the pressure correction is applied. Bold curves show the pointwise median across threshold settings.

weights, so empty tiles avoid both weight reads and matrix multiplication. Applying the same strategy at a and m yields little effective bypass and instead increases full-model latency by 17.2% and 27.4%, respectively. We hypothesize that local sparsity translates into effective bypass only beyond a critical level, where fully zero GPU-friendly tiles occur with high probability.

Attention over q, k, v follows FlashAttention's tiled computation (Dao et al., 2022). In our kernel, sparsity checks are performed only after the operands are loaded, so skipping a matrix instruction saves arithmetic but not operand reads or softmax computation. Moreover, softmax maps zero scores to nonzero probabilities, so sparsity in q, k does not directly propagate to p. Enabling qk/pv skipping together increases full-model latency from 0.514 to 0.521 ms despite substantial bypass. Thus, in our implementation, attention arithmetic savings do not offset the skipping overhead, whereas sparsity at h/z additionally avoids weight reads.

Table 4: **Conditional latency savings**: 14M T_7/P_{all}, $\kappa = 0.5$. Latency is measured per 2,048-token sequence on RTX 5090 (BF16, batch one). Saved time is model latency with one skipping path off minus latency with all paths on. Positive values indicate a benefit (reduced latency). Brackets give [minimum, maximum] savings across all off/on pairs of three independent timing runs. Each run reports the geometric mean over 64 sequences.

Operation	Sparsity (%)	Bypass (%)	Saved time (μs)	Process span (μs)
a: QKV projection	75.6	8.4	-1.6	$[-6.0, +3.8]$
m: FFN-up	68.6	0.0	-2.4	$[-5.1, +2.7]$
h: FFN-down	99.9	94.5	$+150.1$	$[+145.4, +154.4]$
z: attention-output projection	99.9	99.6	$+29.0$	$[+23.7, +35.7]$
q, k: QK scores	93.5 / 94.5	58.0	-2.9	$[-7.3, +1.1]$
v: PV product	98.7	67.2	-6.2	$[-9.0, -2.2]$

4.4 Targeted and Broad Thresholding Across Model Sizes

Figure 5 extends the comparison between targeted T_2/P_h and broad T_7/P_h thresholding to 14M, 31M, and 70M models. Both recipes apply OL1 pressure only at h, isolating the effect of expanding thresholding from h, z to all seven sites.

Across all three model sizes and matched κ, **targeted thresholding preserves quality better than broad thresholding**, with T_2/P_h consistently attaining lower validation loss than T_7/P_h. Broader thresholding, however, reaches lower minimum latency in each sweep, revealing a persistent quality–latency trade-off.

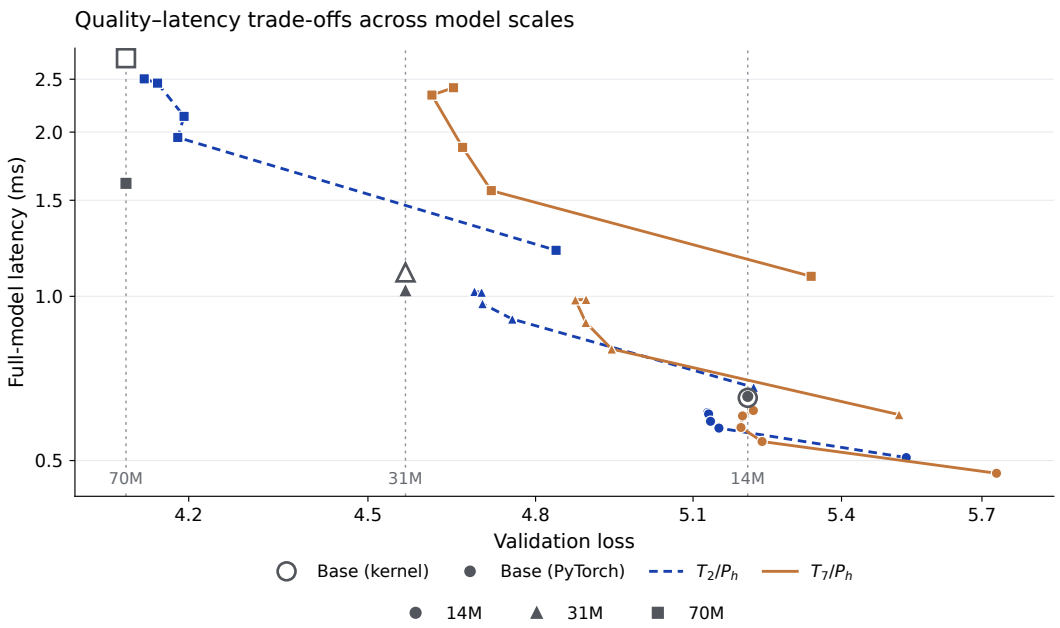

Figure 5: **Quality–latency trade-offs across model scales.** Circles, triangles and squares denote 14M, 31M and 70M. Dashed blue T_2/P_h thresholds h, z; solid orange T_7/P_h thresholds all seven sites. Both apply OL1 pressure only at h. Segments connect measured points in increasing $\kappa \in \{0, 0.01, 0.05, 0.1, 0.5\}$ order. Complete results appear in Appendix D. Both axes are logarithmic. Timings use RTX 5090, BF16, batch one, 2,048 tokens and full vocabulary logits.

Scaling exposes both a systems bottleneck and new operating points. In our current implementation, specialized-kernel overhead grows sharply with model size: relative to native Base execution, it is 8.0% at 31M (1.106 vs. 1.024 ms) and 69.6% at 70M (2.732 vs. 1.611 ms). At the same time, the execution gains from sparsity also grow with scale: increasing κ from 0 to 0.1 under T_2/P_h reduces kernel latency by 0.039 ms at 14M, 0.112 ms at 31M, and 0.549 ms at 70M. These results also expose a limitation of our approach: the kernel was designed around the 14M setting and then scaled to larger models. Its growing overhead suggests that sparse training and kernel design should be co-optimized for each scale, rather than developed independently.

5 Discussion and Conclusion

Our results suggest that effective train-time sparsification should be targeted rather than broad. Different activation sites have different susceptibility to sparsification: in our setting, the FFN hidden activation h and attention-output projection input z respond most favorably, while broader pressure faces greater gradient resistance and conflict. The best trade-offs therefore come from moderate intervention at susceptible sites, rather than maximizing sparsity throughout the model. Simple mechanisms are often sufficient: even fixed-threshold nonlinearities can induce the model to reorganize its representations around the imposed constraint.

More broadly, sparsification is one instance of a training–inference co-design problem. The relevant question is not simply where a model can be made sparse, but which representation makes expensive computation removable at inference time. A systems-guided approach is therefore to identify the operations that dominate latency, determine the structured property that would make them cheaper or skippable, and enforce that property during training. This principle may extend beyond activation sparsity to tiled activation or weight sparsity, quantization, batched execution, and other hardware-dependent forms of structure.

5.1 Failures and Limitations

Our main scaling analysis is limited to models up to 70M parameters. We also trained 410M models, but MiniPile's limited dataset size places them in a different optimization regime: under the same token budget, the 410M Base model attains higher loss than the 70M model and exhibits larger gradient norms (Appendix D.1). We therefore exclude 410M from the primary scaling comparison. Kernel efficiency also does not transfer directly across scales. Adapting the 14M kernel to 70M introduces substantial overhead. Further optimization reduces this overhead while preserving sparse execution, but does not fully close the gap with native PyTorch for dense execution. Our runtime study is deliberately narrow. We use batch-one, full-sequence inference on a single GPU to connect sparsity to execution in a simple, controlled setting rather than to provide a comprehensive production-serving evaluation. Training comparisons use one seed per condition. This limits confidence in individual point estimates, but the consistent trade-off structure across dozens of matched recipes supports the broader trends we report. Finally, the intervention space remains only partially explored. Pressure and threshold placements are combinatorial, and broad-pressure experiments change several pressure targets at the same time. Because OL1 also averages pressure across targeted sites, these comparisons do not isolate the contribution of each added site. Future work could separate pressure targets one at a time, remove this cross-site averaging confound, and allow site-specific thresholds rather than sharing a single κ across sites.

AI use statement

We used OpenAI Codex with GPT-6 Astra to assist with experimental design refinement, mathematical exposition, implementation and debugging of training and evaluation code, GPU-kernel development, experiment execution and monitoring, analysis and interpretation of results, figure preparation, and manuscript drafting and revision. The authors directed the research and approved experimental designs. AI-assisted implementations were checked through tests and comparisons with reference computations; the kernel validation procedures and their limitations are documented in Appendix E. The authors take responsibility for the final text, code, results and scientific claims.

References

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 2397–2430. PMLR, 2023. URL https://proceedings.mlr.press/v202/biderman23a.html.

Edoardo Cetin, Stefano Peluchetti, Emilio Castillo, Akira Naruse, Mana Murakami, and Llion Jones. Sparser, Faster, Lighter Transformer Language Models. *arXiv preprint arXiv:2603.23198*, 2026. URL https://arxiv.org/abs/2603.23198.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv preprint arXiv:2205.14135*, 2022. URL https://arxiv.org/abs/2205.14135.

Yu-Guan Hsieh, James Thornton, Eugene Ndiaye, Michal Klein, Marco Cuturi, and Pierre Ablin. Careful with that Scalpel: Improving Gradient Surgery with an EMA. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 19085–19100. PMLR, 2024. URL https://proceedings.mlr.press/v235/hsieh24a.html.

Jean Kaddour. The MiniPile Challenge for Data-Efficient Language Models. *arXiv preprint arXiv:2304.08442*, 2023. URL https://arxiv.org/abs/2304.08442.

Donghyun Lee, Je-Yong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. CATS: Contextually-Aware Thresholding for Sparsity in Large Language Models. In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=v3w2a7EInO.

James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-Free Activation Sparsity in Large Language Models. *arXiv preprint arXiv:2408.14690*, 2024. URL https://arxiv.org/abs/2408.14690.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1711.05101.

Mao Luo, Hongbin Li, Feng Lin, Hanling Yi, and Zhe Huang. Agentic Kernel Optimization: Generating State-of-the-Art GPU Kernels Without Hand-Written CUDA. *arXiv preprint arXiv:2608.14560*, 2026. URL https://arxiv.org/abs/2608.14560.

Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C. Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. *arXiv preprint arXiv:2310.04564*, 2023. URL https://arxiv.org/abs/2310.04564.

OpenAI. GPT-6 Astra: A New Generation of Intelligence. OpenAI, September 2026. URL https://openai.com/index/gpt-6-astra/. Accessed September 19, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs Write Efficient GPU Kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, and Maosong Sun. ProSparse: Introducing and Enhancing Intrinsic Activation Sparsity within Large Language Models. *arXiv preprint arXiv:2402.13516*, 2024. URL https://arxiv.org/abs/2402.13516.

Hongyu Wang, Shuming Ma, Ruiping Wang, and Furu Wei. Q-Sparse: All Large Language Models can be Fully Sparsely-Activated. *arXiv preprint arXiv:2407.10969*, 2024. URL https://arxiv.org/abs/2407.10969.

Chong You, Kan Wu, Zhipeng Jia, Lin Chen, Srinadh Bhojanapalli, Jiaxian Guo, Utku Evci, Jan Wassenberg, Praneeth Netrapalli, Jeremiah J. Willcock, Suvinay Subramanian, Felix Chern, Alek Andreev, Shreya Pathak, Felix Yu, Prateek Jain, David E. Culler, Henry M. Levy, and Sanjiv Kumar. Spark Transformer: Reactivating Sparsity in FFN and Attention. *arXiv preprint arXiv:2506.06644*, 2025. URL https://arxiv.org/abs/2506.06644.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient Surgery for Multi-Task Learning. In *Advances in Neural Information Processing Systems*, 2020. URL https://arxiv.org/abs/2001.06782.

A ORTHOGONAL L1 (OL1)

Objective and update. For a nonempty pressure set $\mathcal{P}$ and input microbatch x, let $A_j(\theta; x)$, $j = 1, \ldots, J$, denote the activation tensors at the targeted sites and layers, after any thresholding. Let $n_j > 0$ be the total number of entries in A_j, including zeros. Then, define the per-microbatch pressure loss

$$\ell_1(\theta; \mathcal{P}, x) = \frac{1}{J} \sum_{j=1}^{J} \frac{1}{n_j} \sum_{i=1}^{n_j} |A_{j,i}(\theta; x)|. \tag{1}$$

For unpressured recipes, set $\ell_1(\theta; \varnothing) = 0$. Each tensor's mean absolute activation receives equal weight, so adding pressure sites changes both the targets and the averaging normalization.

Standard L1 regularization applies AdamW (Loshchilov & Hutter, 2019) to the globally clipped gradient of $\mathcal{L}_{\text{task}} + \lambda \ell_1$. OL1 instead separates task and pressure updates: task-only AdamW is followed by a pressure correction that reduces the component opposing the adaptive task direction and imposes a relative norm budget. The pressure gradient therefore does not directly enter Adam's moment estimates.

At each optimizer step, both gradients are evaluated at the same pre-update parameters and averaged over the same microbatches. The averaged task gradient is globally clipped before updating AdamW's moments; $\widehat{m}_t$ and $\widehat{v}_t$ denote their bias-corrected values after this update. The pressure gradient g_1 is the average of $\nabla_\theta \ell_1$ over these microbatches and is neither clipped nor momentum-averaged.

The pressure correction is computed over parameter tensors for which both gradients are present and Adam's state is initialized. The vectors below are restricted to these tensors, and norms and inner products are computed over their concatenation. Parameters outside this set receive only the task-only AdamW update. Suppressing the step index on the update directions, define

$$u = \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon_A}, \qquad w = \frac{g_1}{\sqrt{\widehat{v}_t} + \epsilon_A}, \tag{2}$$

where square roots and divisions are elementwise and $\epsilon_A > 0$ is Adam's numerical stabilizer.

For $c = \langle u, w \rangle$, $q = \|u\|_2^2$, and an OL1 stabilizer $\epsilon > 0$, define

$$\widetilde{w} = \begin{cases} w - \dfrac{c}{q + \epsilon} u, & c < 0 \text{ and } q > \epsilon, \\ w, & \text{otherwise.} \end{cases} \tag{3}$$

The projection leaves nonconflicting directions unchanged and is skipped when $q \leq \epsilon$. When the projection branch applies, $\langle u, \widetilde{w} \rangle = c\epsilon/(q + \epsilon)$: the stabilizer reduces, rather than exactly eliminates, opposition. Without the stabilizer, the conflicting component is removed exactly.

For a pressure coefficient $\lambda \geq 0$ and relative norm budget $b > 0$, define

$$r = \frac{\lambda \|\widetilde{w}\|_2}{\|u\|_2 + \epsilon}, \qquad s = \begin{cases} \min\left\{1, \dfrac{b}{r + \epsilon}\right\}, & r > 0, \\ 1, & r = 0. \end{cases} \tag{4}$$

After the task-only AdamW update, each parameter group p receives

$$\theta_p \leftarrow \theta_p - \alpha_p \lambda s \widetilde{w}_p, \tag{5}$$

where α_p is the group's learning rate and the pressure correction is zero outside the participating parameter set. The pressure gradient is not recomputed after the task update.

Before learning-rate scaling, ignoring ϵ and assuming $\|u\|_2 > 0$, the correction norm is

$$\|\lambda s \widetilde{w}\|_2 = \min\{\lambda \|\widetilde{w}\|_2,\, b\|u\|_2\}\,. \tag{6}$$

Thus, holding u and $\widetilde{w}$ fixed, increasing λ has no further effect once the cap is active.

L1 versus OL1. At 14M, we run a first ablation comparing L1 and OL1 with T_1/P_1. Both methods yield similar final metrics under matched initialization, data order, and training budget (Table 5). The absolute difference in model-wide sparsity is below 0.02 percentage points at every tested λ. OL1 has slightly lower validation loss at $\lambda \in \{0.05, 0.1, 0.5\}$ and higher validation loss at $\lambda = 1$.

Table 5: **L1 and OL1 at T_1/P_1 on Pythia-14M.** Final validation loss and model-wide sparsity after 712 optimizer updates under matched initialization and data ordering. Each condition uses one training seed.

	L1		OL1	
λ	S_{model} (%)	Val. loss	S_{model} (%)	Val. loss
0.05	3.1416	5.2061	3.1452	5.1981
0.1	3.3363	5.1655	3.3386	5.1594
0.5	3.7876	5.1127	3.7680	5.1102
1	3.9493	5.1023	3.9384	5.1210

B SPARSITY METRIC

Activation sparsity at a site is the fraction of entries that are exactly zero. We count zeros and total entries across layers and validation sequences before taking their ratio. We define model-wide sparsity by counting scalar multiplications with at least one zero activation operand

$$\mathcal{S}_{\text{model}} = \frac{N_0}{N_{\text{model}}}. \tag{7}$$

Here N_0 counts such multiplications in Pythia-block matrix products, whereas N_{model} counts all scalar multiplications in those products and the final vocabulary projection. Since its not being sparsified, the final vocabulary projection contributes only to the denominator. Counts are summed across blocks and validation sequences.

For a block's linear projection from d_{in} to d_{out} features over n rows, the denominator contribution is $nd_{\text{in}}d_{\text{out}}$. Each zero input entry contributes d_{out} to N_0, corresponding to one multiplication per output feature. Weights zeros doesn't contributes to N_0.

In attention, both operands are activations. We count each multiplication once if either or both operands are zero. For queries q, keys k, values v, and attention probabilities p, one head in one layer and one sequence contributes

$$N_{0,\text{QK}} = \sum_{t=1}^{T}\sum_{u=1}^{t}\sum_{j=1}^{d_h} \mathbf{1}\{q_{tj} = 0 \ \vee \ k_{uj} = 0\}, \tag{8}$$

$$N_{0,\text{PV}} = \sum_{t=1}^{T}\sum_{u=1}^{t}\sum_{j=1}^{d_h} \mathbf{1}\{p_{tu} = 0 \ \vee \ v_{uj} = 0\}. \tag{9}$$

Here T is the sequence length, d_h is the head dimension, t indexes query positions, and u indexes key/value positions. We restrict attention pairs to $u \le t$ in both N_0 and N_{model}, excluding masked future positions. Each attention product therefore contributes $d_h T(T+1)/2$ to the denominator per head, layer, and sequence. We sum these counts over heads, layers, and validation sequences, using the actual matrix-product operands after thresholding and any intervening transformations.

Only scalar multiplications in the specified matrix products are counted. Normalization, softmax, activation functions, additions, and memory transfers are excluded.

B.1 MODEL-WIDE SPARSITY CEILING

To quantify the architectural reach of the thresholding sites $\mathcal{N}$, assume that all activations at these sites are zero. Let $N_{\text{reach}}(\mathcal{N})$ count the scalar multiplications that are structurally guaranteed to have at least one zero activation operand under this assumption, restricted to the same Pythia-block matrix products used to define N_0. The model-wide sparsity ceiling is

$$\mathcal{S}^{\max}_{\text{model}} = \frac{N_{\text{reach}}(\mathcal{N})}{N_{\text{model}}}. \tag{10}$$

The denominator is unchanged. The numerator includes multiplications affected by structurally forced downstream zeros: setting $v = 0$ also makes $z = 0$, affecting both PV and the attention-output projection. Zero attention scores do not imply zero softmax probabilities at unmasked positions.

For fixed-length sequences, the ceiling depends only on the architecture, selected sites, and sequence length. Appendix C.1 gives the counts for each Pythia size.

C EXPERIMENTAL DESIGN

Table 6 gives the shapes and positions of the activation sites in Figure 2. Here T is the sequence length, d is the hidden width, d_f is the FFN width, and H is the number of attention heads, each with dimension $d_h = d/H$. Each intervention uses the same thresholding set $\mathcal{N}$ and pressure set $\mathcal{P}$ in every layer, with a single threshold κ across all selected thresholding sites.

Table 6: Activation shapes and intervention positions for one sequence.

Site	Shape	Following matrix product	Position
a	$T \times d$	QKV projection	After attention LayerNorm
m	$T \times d$	FFN-up projection	After FFN LayerNorm
h	$T \times d_f$	FFN-down projection	Activation-function output
q, k	$H \times T \times d_h$	Attention scores	After RoPE
v	$H \times T \times d_h$	Probability–value product	After the QKV split
z	$T \times d$	Attention-output projection	Concatenated attention context

C.1 ARCHITECTURE-SPECIFIC SPARSITY CEILINGS

Under the accounting convention in Appendix B, the matrix products in one Pythia block contribute, per sequence, $3Td^2$ scalar multiplications for the QKV projection, Td^2 for the attention-output projection, Tdd_f for each FFN projection, and $dT(T+1)/2$ for each of $QK^\top$ and PV. The attention counts include only unmasked causal pairs and are summed over all heads. With L layers and vocabulary size V_{vocab}, the per-sequence totals are

$$\begin{aligned} C_{\text{block}} &= T(4d^2 + 2dd_f) + dT(T+1), \\ C_{\text{model}} &= LC_{\text{block}} + TdV_{\text{vocab}}. \end{aligned} \tag{11}$$

Let $C_{\text{reach}}(\mathcal{N})$ denote the per-block contribution to the ceiling numerator defined in Appendix B.1. All configurations studied here use $d_f = 4d$, giving $C_{\text{reach}}(\mathcal{N}) = 0$ for T_0, $4Td^2$ for T_1, $5Td^2$ for T_2, $12Td^2$ for T_4, and C_{block} for T_7. In particular, T_2 covers the FFN-down and attention-output projections, whose contributions sum to $Tdd_f + Td^2 = 5Td^2$. Because all counted sequences have the same length, their number cancels from the ceiling ratio:

$$\mathcal{S}^{\max}_{\text{model}} = \frac{L\, C_{\text{reach}}(\mathcal{N})}{C_{\text{model}}}. \tag{12}$$

Table 1 uses $T = 2{,}048$, $V_{\text{vocab}} = 50{,}304$, and $(L, d) = (6, 128), (6, 256), (6, 512), (24, 1{,}024)$ for 14M, 31M, 70M, and 410M, respectively.[1]

C.2 TRAINING AND EVALUATION

All model sizes use the Pythia-14M-deduped tokenizer. The initialization and data-order seeds are both set to 1234, with one training seed per condition. All runs use AdamW with $(\beta_1, \beta_2) = (0.9, 0.95)$, $\epsilon_A = 10^{-8}$, and weight decay 0.1. Global ℓ_2-norm gradient clipping uses a threshold of 1 and is applied as described in Appendix A. Attention and hidden dropout probabilities are both zero.

Each run uses 712 optimizer updates, with 1% linear learning-rate warmup followed by cosine decay. Table 7 gives the peak and minimum learning rates, microbatch sizes, and common effective batch size. Gradients are averaged over the listed number of microbatches before each optimizer update. With 1,024 sequences per update and 2,048 tokens per sequence, each run processes 1,493,172,224 input tokens.

[1] Pinned architecture configurations: 14M, 31M, 70M, and 410M.

Table 7: **Training settings across model sizes.** The effective batch size is the microbatch size multiplied by the number of gradient accumulation steps. The common input-token budget gives fewer tokens per parameter for larger models.

Setting	14M	31M	70M	410M
Parameters	14,067,712	30,494,720	70,426,624	405,334,016
Peak learning rate	10^{-3}	10^{-3}	10^{-3}	3×10^{-4}
Minimum learning rate	10^{-4}	10^{-4}	10^{-4}	3×10^{-5}
Microbatch size (sequences)	32	4	4	4
Gradient accumulation steps	32	256	256	256
Effective batch size (sequences/update)	1,024	1,024	1,024	1,024
Sequence length	2,048	2,048	2,048	2,048
Tokens per parameter	106.14	48.96	21.20	3.68

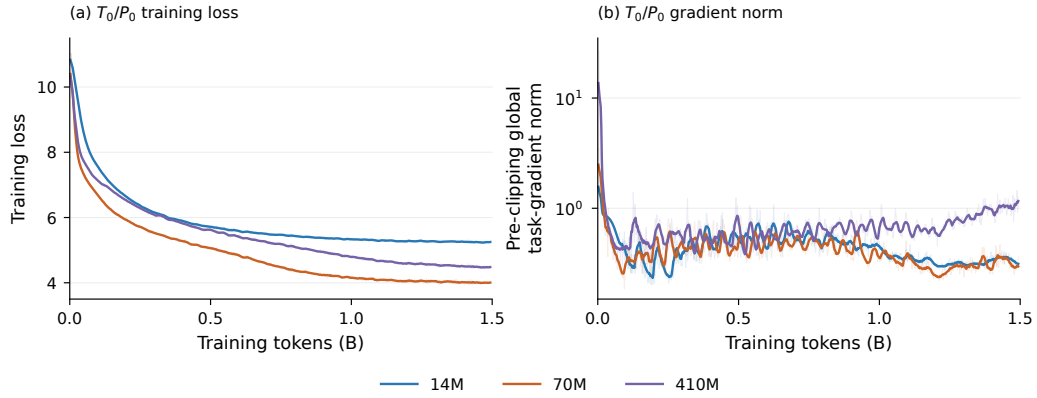

Figure 6: **Base-model (T_0/P_0) training at 14M, 70M, and 410M. (a)** Training loss and **(b)** the global task-gradient norm before clipping, shown on a logarithmic vertical axis. Faint lines show per-update values; solid lines show centered nine-update moving averages, with shorter windows near the endpoints. Gradient norms are computed after gradient accumulation and are not normalized by parameter count.

D Detailed results

Table 8 compares the recipes evaluated at both 14M and 70M. Loss and model-wide sparsity use the same definitions and final checkpoints as the main figures, with one training seed and complete validation per condition. Table 9 adds the 31M Base and matched T_2/P_h, T_7/P_h sweeps.

Table 8: **14M/70M results.** Parentheses give changes in loss and latency relative to each size's T_0/P_0 Base. Base uses dense PyTorch/SDPA; other rows use the optimized implementation for their size. Latency is in ms (RTX 5090, BF16, batch one, 2,048 tokens, full logits). OL1 uses $\lambda = b = 1$.

Recipe	κ	14M			70M		
		$\mathcal{S}_{\text{model}}$ (%)	Loss (Δ)	Latency (Δ)	$\mathcal{S}_{\text{model}}$ (%)	Loss (Δ)	Latency (Δ)
T_0/P_0 (Base)	—	0.00	5.21 (+0.000)	0.66 (+0.000)	0.00	4.10 (+0.000)	1.61 (+0.000)
T_1/P_0 (ReLU)	—	2.71	5.27 (+0.061)	0.63 (-0.023)	10.07	4.22 (+0.123)	2.72 (+1.106)
T_2/P_h	0	4.63	5.13 (-0.080)	0.61 (-0.043)	14.63	4.13 (+0.029)	2.50 (+0.893)
	0.01	4.72	5.13 (-0.078)	0.61 (-0.047)	14.70	4.15 (+0.050)	2.46 (+0.848)
	0.05	5.00	5.13 (-0.074)	0.59 (-0.065)	15.03	4.19 (+0.093)	2.14 (+0.526)
	0.1	5.17	5.15 (-0.057)	0.57 (-0.082)	15.21	4.18 (+0.083)	1.96 (+0.345)
	0.5	5.34	5.54 (+0.328)	0.51 (-0.149)	15.43	4.84 (+0.738)	1.21 (-0.397)

Recipe	κ	14M			70M		
		$\mathcal{S}_{\text{model}}$ (%)	Loss (Δ)	Latency (Δ)	$\mathcal{S}_{\text{model}}$ (%)	Loss (Δ)	Latency (Δ)
T_4/P_h	0	8.41	5.22 (+0.007)	0.60 (-0.052)	25.79	4.61 (+0.507)	2.38 (+0.772)
	0.01	8.59	5.20 (-0.004)	0.59 (-0.063)	25.97	4.62 (+0.519)	2.37 (+0.754)
	0.05	9.04	5.20 (-0.013)	0.56 (-0.094)	26.57	4.60 (+0.497)	1.89 (+0.281)
	0.1	9.34	5.23 (+0.020)	0.53 (-0.126)	27.22	4.70 (+0.602)	1.57 (-0.038)
	0.5	10.23	5.72 (+0.514)	0.46 (-0.193)	29.24	5.31 (+1.206)	1.09 (-0.523)
T_4/P_{all}	0	7.81	5.46 (+0.250)	0.63 (-0.022)	25.67	4.81 (+0.706)	2.60 (+0.988)
	0.01	8.59	5.46 (+0.250)	0.62 (-0.035)	26.22	4.82 (+0.723)	2.50 (+0.886)
	0.05	10.46	5.49 (+0.281)	0.57 (-0.083)	28.49	4.87 (+0.774)	2.02 (+0.412)
	0.1	11.39	5.55 (+0.340)	0.53 (-0.122)	30.57	4.94 (+0.844)	1.69 (+0.076)
	0.5	12.71	6.04 (+0.829)	0.46 (-0.196)	35.60	5.39 (+1.290)	1.10 (-0.506)
T_7/P_h	0	8.40	5.22 (+0.011)	0.62 (-0.038)	25.79	4.65 (+0.551)	2.41 (+0.801)
	0.01	8.86	5.20 (-0.010)	0.60 (-0.052)	26.01	4.61 (+0.512)	2.34 (+0.727)
	0.05	10.13	5.19 (-0.014)	0.57 (-0.081)	26.89	4.67 (+0.567)	1.88 (+0.264)
	0.1	10.90	5.24 (+0.029)	0.54 (-0.114)	27.78	4.72 (+0.619)	1.56 (-0.049)
	0.5	16.66	5.73 (+0.523)	0.47 (-0.182)	32.23	5.34 (+1.237)	1.09 (-0.523)
T_7/P_{all}	0	7.05	5.48 (+0.272)	0.63 (-0.021)	23.56	4.94 (+0.841)	2.74 (+1.130)
	0.01	7.72	5.48 (+0.267)	0.64 (-0.017)	24.23	4.94 (+0.837)	2.73 (+1.121)
	0.05	9.86	5.46 (+0.254)	0.63 (-0.021)	26.52	4.96 (+0.864)	2.72 (+1.106)
	0.1	11.80	5.43 (+0.221)	0.59 (-0.062)	28.69	4.95 (+0.850)	2.68 (+1.067)
	0.5	27.48	5.83 (+0.621)	0.47 (-0.182)	40.60	5.22 (+1.116)	1.28 (-0.335)

Table 9: **31M results.** Base uses native PyTorch; T_2/P_h and T_7/P_h use the specialized implementation, with OL1 only at h and $\lambda = b = 1$.

Recipe	κ	$\mathcal{S}_{\text{model}}$ (%)	Loss	Δ loss	Latency (ms)	Δ latency (ms)
T_0/P_0 (Base)	—	0.00	4.566	+0.000	1.024	+0.000
T_2/P_h	0	8.59	4.688	+0.123	1.021	-0.003
T_2/P_h	0.01	8.73	4.701	+0.136	1.017	-0.007
T_2/P_h	0.05	9.06	4.703	+0.137	0.969	-0.054
T_2/P_h	0.1	9.26	4.757	+0.192	0.908	-0.115
T_2/P_h	0.5	9.47	5.220	+0.655	0.681	-0.343
T_7/P_h	0	15.34	4.894	+0.329	0.987	-0.037
T_7/P_h	0.01	15.74	4.874	+0.309	0.985	-0.039
T_7/P_h	0.05	16.77	4.894	+0.328	0.896	-0.128
T_7/P_h	0.1	17.53	4.943	+0.377	0.801	-0.223
T_7/P_h	0.5	22.66	5.521	+0.955	0.607	-0.417

D.1 410M RESULTS

Table 10 reports all twelve trained 410M conditions. The 410M base-model loss exceeds 70M's 4.100. All sizes receive 712 optimizer steps and 1.493B input tokens, corresponding to 106.14, 48.96, 21.20 and 3.68 tokens per parameter at 14M, 31M, 70M and 410M. The 410M schedule also has a lower peak learning rate (3×10^{-4} rather than 10^{-3}). The training trajectories are retained in Figure 6. The main runtime comparison uses 14M, 31M and 70M.

Table 10: **410M results.** All twelve trained conditions; pressure uses OL1 with $\lambda = b = 1$. Ceiling is $100\mathcal{S}^{\max}_{\text{model}}$ at 2,048 tokens. This size has no P_h or unpressured T_4/T_7 runs. The 410M model has not been tested under specialized kernel execution.

Recipe	κ	Loss	Ceiling (%)	$\mathcal{S}_{\text{model}}$ (%)
T_0/P_0 (Base)	—	4.5475	0.00	0.011
T_1/P_0 (ReLU)	—	4.6513	24.93	21.050
T_4/P_{all}	0	5.6921	74.78	38.339
	0.01	5.6914		39.149
	0.05	5.6789		42.125
	0.1	5.6789		45.688
	0.5	5.1911		71.591
T_7/P_{all}	0	5.4263	87.25	41.121
	0.01	5.4331		41.478
	0.05	5.4679		44.053
	0.1	5.4908		48.998
	0.5	5.1209		80.616

E KERNEL IMPLEMENTATION AND VALIDATION

E.1 SPARSE EXECUTION

The specialized implementations maintain the trained weights and thresholding operators and exploit the resulting zero activations. Kernel development proceeded iteratively: we first developed and optimized the 14M implementation, adapted it to the 70M model, performed a second round of optimization, and finally adapted the optimized design to the 31M model. At h, z, activation tiles (i.e. small submatrices of activations processed together on the GPU) are checked before the associated weights are loaded. Empty tiles can therefore bypass both weight reads and matrix instructions, while sufficiently sparse rows can be handled with scalar operations over only the nonzero positions instead of issuing a dense matrix instruction. By contrast, at a, m, q, k, v, the 14M and initial 70M kernels check for zero activations only after loading the operands. So, zero blocks can skip the matrix instruction, but cannot avoid the associated memory reads. Table 11 summarizes these execution paths, and the supplementary code provides the full implementations.

Table 11: **Sparse execution mechanisms in the 14M and initial 70M implementations.** MMA denotes a matrix multiply–accumulate instruction. Tile dimensions are given as tokens by input features. All sparsity checks are performed during the forward pass. Later implementation changes are described below.

Operation	Execution rule	Work avoided
QKV, FFN-up (a, m)	Check activation blocks after both operands have been loaded.	Empty blocks skip MMA instructions, but the associated operand reads are retained.
FFN-down (h), attention-output (z)	Check 8×16 activation tiles before loading the corresponding weights. Rows with at most two nonzero activations switch to scalar operations over the nonzero positions when numerical guards permit; all other rows follow the matrix path.	Empty tiles avoid both weight reads and MMA instructions. Scalar-path rows read only the weights associated with nonzero activations.
Attention ($QK^\top, PV$)	Check operand blocks after both operands have been loaded.	Skip MMA instructions when either block is zero, while retaining operand reads, masking, and softmax.

Adaptation across scales. The execution rules in Table 11 were designed iteratively around the post-hoc sparsity structure of the trained representations, with the explicit goal of reducing inference

latency. This process motivates our train–inference co-design perspective: the inference system reveals which forms of sparsity are practically exploitable and, in turn, which structures training should preferentially induce. The larger kernels are adaptations of the 14M implementation rather than independently optimized designs. At 70M, a subsequent optimization round reduces overhead but leaves Base latency at 2.732 ms, compared with 1.611 ms for the native implementation. The 31M implementation adapts the optimized 70M policy without a new optimization search: input projections and attention use dense execution, while h, z retain empty-tile skipping, scalar execution over sufficiently sparse rows, and a matrix-instruction fallback.

E.2 Numerical validation and timing protocol

Measurements use an RTX 5090, BF16, batch size 1, 2,048-token causal sequences, and full-vocabulary logits at every token, with PyTorch 2.11.0, Transformers 5.12.1, and CUDA 12.8. We evaluate full-sequence inference with fixed weights and tensor shapes. Cached decoding and concurrent serving are outside the scope of this evaluation.

Final checkpoints are validated against the standard PyTorch implementation using the same weights and thresholding operators. Validation is run in three independent processes over all 338 validation sequences. For specialized logits y and reference logits $\hat{y}$, acceptance requires finite outputs, $|y-\hat{y}| \leq 0.25+0.02|\hat{y}|$ for every logit, relative ℓ_2 error of at most 0.02 per sequence, and an absolute difference in mean validation loss of at most 0.001. All final checkpoints pass these criteria.

PyTorch/SDPA and specialized implementations use CUDA graphs with identical inputs. We report geometric means of synchronized host timings over 64 fixed validation sequences, using seven randomized paired passes in each of three independent processes. Timings include embeddings, transformer blocks, final normalization, vocabulary logits, thresholding, and all input-dependent sparsity checks. Compilation, static weight preparation, graph capture, and input staging are excluded.