MIXTURE OF KV GROUPS: TRADING KV-CACHE CAPACITY FOR DECODE BANDWIDTH

Anonymous authors
Paper under double-blind review

ABSTRACT

KV cache memory bandwidth has emerged as a major bottleneck to LLM inference and RL training, especially at long contexts. Grouped-query attention (GQA) has become a widely used method to accelerate decoding by reducing KV-cache traffic, but aggressive KV sharing can impair long-context quality. We study Mixture of KV Groups, which expands KV capacity while routing each token to a small subset of KV groups. Every group retains the complete sequence history, trading additional cache storage for sparse KV memory access. We train 220M-class and 3B hybrid language models from scratch, jointly learning KV representations and routing. At 220M scale and 3B parameter scales we find that Mixture of KV Groups reliably improves on the long context performance of GQA models while preserving much of their decode efficiency. We also develop an attention masking scheme to enable parallel sparse training with FlexAttention.

1 INTRODUCTION

At long contexts repeated high bandwidth KV memory reads quickly become the bottleneck in language-model decoding. Grouped-query attention (GQA) (Ainslie et al., 2023) accelerates decoding by sharing keys and values across query heads, reducing both the stored cache and the amount read at each step. This saves bandwidth but also reduces the number of distinct representations of the sequence available to attention, which can impair long-context quality (Bertsch et al., 2026). We study how to improve long-context quality while retaining the decode advantage of GQA models with few KV groups.

We study Mixture of KV Groups, which gives attention a larger set of KV groups and learns to select a small subset for each token. Each group stores the complete sequence history, but a token reads only its selected groups. This separates how much information the model stores from how much it accesses at each decoding step: additional KV capacity need not entail proportionally more reads. The cost is a larger cache and additional work to maintain it.

Prior work has explored token-dependent attention routing (Zhang et al., 2022; Jin et al., 2024), and Polar Sparsity (Shrestha et al., 2025) demonstrates selective KV access for faster decoding in pretrained dense models. We investigate increasing total KV capacity while holding the number of active groups fixed. We train models from scratch so that the KV representations and routing policy can adapt jointly to this design throughout training. Section 5 discusses the distinctions from prior routing methods.

Experiments with 220M and 3B hybrid language models show that sparse routing can substantially improve long-context quality while preserving most of the decode advantage of aggressive GQA. At 220M scale, sparse models occupy all but the fastest point of the measured quality–latency Pareto frontier (Figure 1). The sparse 3B model achieves a LongPPL of 4.59 compared with dense KV8's 4.58, while delivering 1.60–$1.94\times$ its peak aggregate decode throughput. Compared with dense KV4, it improves LongPPL from 4.71 to 4.59 while retaining 86–95% of peak aggregate throughput across the tested context lengths.

We also develop a parallel training implementation using FlexAttention (Guessous et al., 2024) that executes the selected attention computations without custom GPU kernels. We measure its throughput relative to dense attention and quantify the masked computation introduced by tiled execution to characterize the attention-operator cost of parallel sparse training.

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.

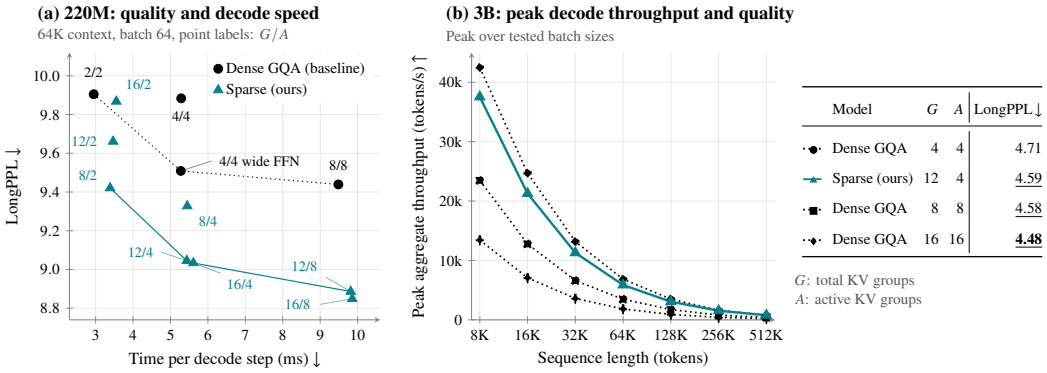

Figure 1: **Sparse routing improves the quality–decode-efficiency tradeoff.** LongPPL is evaluated on sequences of up to 32K tokens. **Left:** At 220M scale, sparse models occupy all but the fastest point of the measured quality–latency Pareto frontier. Lower is better on both axes. Connecting lines show the separate dense and sparse observed Pareto frontiers and serve as visual guides. **Right:** The sparse 3B model ($G = 12$, $A = 4$) nearly matches dense KV8's LongPPL (4.59 versus 4.58), while achieving 1.60–$1.94\times$ its peak aggregate decode throughput across the tested context lengths. It also improves on dense KV4's LongPPL (4.71), retaining 86–95% of its peak throughput. G and A denote total and active KV groups.

2 Preliminaries

2.1 Self-Attention

For an input sequence $X \in \mathbb{R}^{n\times d}$, a single self-attention head with weights $W_Q, W_K, W_V, W_O \in \mathbb{R}^{d\times d_h}$ computes $\mathrm{Attn}(XW_Q, XW_K, XW_V)W_O^\top$, where

$$\mathrm{Attn}(Q, K, V) = \mathrm{Softmax}\left(\frac{QK^\top}{\sqrt{d_h}} + M\right)V. \tag{1}$$

Here d_h denotes the head dimension and $M_{ij} = 0$ for $j \le i$ and $-\infty$ otherwise enforces causality for the full sequence. For a single query attending only to its causal prefix, $M = 0$. Multi-head attention (MHA) sums the outputs of multiple heads. Self-attention can be computed efficiently in parallel during training and prefill using FlashAttention (Dao et al., 2022). During autoregressive decoding, Q has only one row, and attention is typically memory-bandwidth bound.

2.2 Grouped-Query Attention

Grouped-query attention (GQA) (Ainslie et al., 2023) splits H query heads into G equally sized groups with each group of H/G query heads sharing key and value projections W_K, W_V while retaining separate query and output projections W_Q^h, W_O^h. The output of a single group is

$$\sum_{h=1}^{H/G} \mathrm{Attn}(XW_Q^h, XW_K, XW_V)(W_O^h)^\top. \tag{2}$$

The full layer sums the outputs of all G groups. Sharing keys and values reduces KV cache size by a factor of H/G relative to MHA with the same query-head count but maintains the same attention-product FLOPs and approximately the same training cost.

2.3 Mixture of Experts

Sparse mixture-of-experts (MoE) layers increase parameter capacity by activating only a subset of feedforward networks, called experts, for each token (Shazeer et al., 2017). For a token representation $x \in \mathbb{R}^{1\times d}$, consider E experts $f_\ell : \mathbb{R}^{1\times d} \to \mathbb{R}^{1\times d}$. We consider top-$k$ routing with sigmoid gates. A bias-free linear router with weights $W_r \in \mathbb{R}^{d\times E}$ computes

$$s(x) = \sigma(xW_r), \tag{3}$$

where σ is the elementwise sigmoid function. Given expert-specific selection biases $b \in \mathbb{R}^E$, the selected experts and layer output are

$$\mathcal{S}(x) = \mathrm{TopK}(s(x) + b, k), \tag{4}$$

$$\mathrm{MoE}(x) = \sum_{\ell \in \mathcal{S}(x)} \frac{s_\ell(x)}{\sum_{r \in \mathcal{S}(x)} s_r(x)} f_\ell(x). \tag{5}$$

The selection biases affect which experts are activated, but not their output weights. The feedforward MoE layers normalize the selected scores; our attention layers use them without renormalization. At fixed expert size and k, increasing E increases total expert parameters without increasing the number of expert evaluations per token, although routing and communication still incur overhead.

Unconstrained routing can collapse onto a few experts: frequently selected experts receive more training, reinforcing their selection while other experts remain undertrained (Shazeer et al., 2017). Uneven loads also create bottlenecks when experts are distributed across devices. Auxiliary balancing losses encourage more uniform expert utilization (Fedus et al., 2022). Alternatively, auxiliary-loss-free balancing adjusts the selection biases using observed token counts (Wang et al., 2024):

$$b_\ell \leftarrow b_\ell + \eta \, \mathrm{sign}(\bar{c} - c_\ell), \qquad \bar{c} = \frac{1}{E} \sum_{r=1}^{E} c_r, \tag{6}$$

where c_ℓ counts assignments to expert ℓ over the update interval and η is the bias update rate. This increases the selection biases of underused experts and decreases those of overused experts. The biases are not learned through backpropagation, and no load-balancing auxiliary loss is required.

3 Mixture of KV Groups

3.1 Architecture

For an input sequence $X \in \mathbb{R}^{n \times d}$, a Mixture of KV Groups layer contains G KV groups and H query heads, with H/G query heads per group. Figure 2 compares dense GQA with this sparse layer.

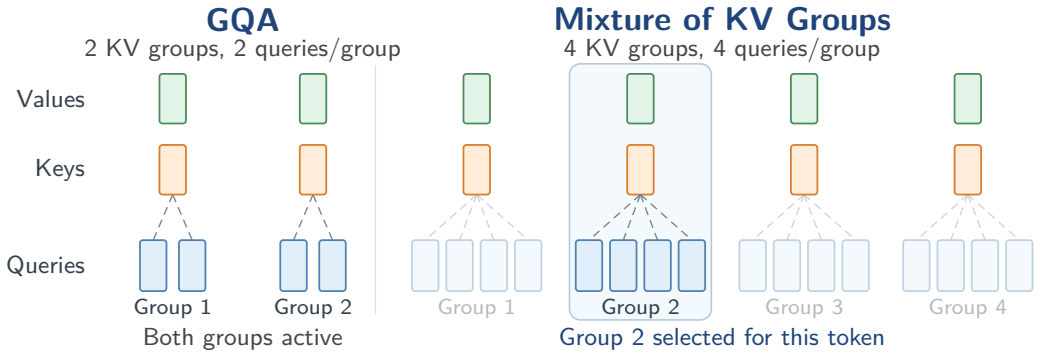

Figure 2: Dense GQA and Mixture of KV Groups for one token. Left: GQA uses two KV groups with two query heads per group. Right: the sparse layer contains four KV groups with four query heads per group and selects one group for this token ($G = 4$, $A = 1$). Both examples activate four query heads. Dimmed queries and connections indicate unselected groups; K/V projections and cache writes remain dense across all groups.

We compute key and value projections for every token in every group:

$$K_\ell = XW_K^\ell, \qquad V_\ell = XW_V^\ell, \qquad \ell = 1, \ldots, G. \tag{7}$$

These projections are independent of routing, so each group retains the complete causal KV history.

For each token $x_i = X_{i,:}$, the sigmoid-gated router described in Section 2.3 selects $A \leq G$ groups:

$$s_i = \sigma(x_i W_r), \qquad \mathcal{S}_i = \mathrm{TopK}(s_i + b, A), \tag{8}$$

where $W_r \in \mathbb{R}^{d\times G}$ is a bias-free linear projection and $b \in \mathbb{R}^G$ contains the non-trainable selection biases. The token output is a weighted sum of the selected groups' outputs:

$$y_i = \sum_{\ell\in\mathcal{S}_i} s_{i,\ell} \sum_{h=1}^{H/G} \mathrm{Attn}\Big(x_i W_Q^{\ell,h}, K_{\ell,1:i}, V_{\ell,1:i}\Big)\,(W_O^{\ell,h})^\top. \tag{9}$$

The selection biases affect only group selection; the output weights are the original sigmoid scores without renormalization.

Active and total parameters. Compared with dense GQA with A KV groups and the same number of query heads per group, our sparse layer uses the same number of query and output projection parameters per token. The additional active parameters are the router weights and the K/V projection weights for the $G - A$ extra groups, which are used for every token. In the configurations we study, these additions are small relative to the model's overall active parameter count. Total parameters can increase more substantially: every group retains its own query and output projections, even when it is not selected. Thus, configurations with a large ratio of total to active groups (G/A) can have substantially more total parameters while maintaining nearly matched active parameter counts.

KV capacity and decode efficiency. This design separates stored KV capacity, determined by the total group count G, from decode-time KV reads, determined by the active count A. The larger cache reduces the batch size that fits in memory, so the benefit of fewer reads must be weighed against reduced weight amortization.

3.2 Parallel Training

We develop a parallel training implementation using FlexAttention (Guessous et al., 2024) with the FlashAttention-4 backend (Zadouri et al., 2026). The central challenge is that groups receive different numbers of queries, while each group retains the complete KV history. We address this by concatenating queries across groups and constructing a mask that enforces KV-group assignments, causal sequence order, and document boundaries. Figure 3 illustrates routing, packing, masking, and tile execution.

Each batch element contains n tokens from one or more document segments. For each token assigned to a KV group, that group's H/G query vectors occupy separate rows. We concatenate these rows by group along the query sequence axis and concatenate all groups' K/V histories along the key/value sequence axis. We call this arrangement query packing. FlexAttention receives $nA(H/G)$ query rows and nG key/value rows per batch element, with its head axis set to size one. Groups may receive different numbers of queries, but their combined query count is fixed.

For each query row, we retain its KV group, original token position, and the start position of its document. The mask allows attention only to keys in the query's KV group and document whose sequence positions are at or before the query's original position. Causal masking uses the original sequence positions, not the row indices after packing. FlexAttention processes rectangular blocks of query–key interactions, called tiles. From each query's allowed key range, we construct lists of the tiles containing allowed interactions, without materializing a token-level attention mask. Tiles with no allowed interactions are skipped. Within the remaining tiles, matrix products may also compute disallowed scores, which are excluded from the softmax. Appendix A provides the implementation, including query packing, block-mask construction, and output reordering.

4 Experiments

4.1 Experimental Setup

Model training setup. We train two families of hybrid language models combining Gated DeltaNet (GDN) layers (Yang et al., 2024) with full-context self-attention layers. In the attention layers, we compare dense GQA with Mixture of KV Groups. We refer to the smaller family as 220M, although its configurations range from approximately 213M to 268M parameters. This family includes 11 standard attention configurations and a twelfth configuration: a dense model with four KV groups



Figure 3: Routing and parallel attention execution for an example layer with two total KV heads and one active KV head per token ($G = 2$, $A = 1$). (a) Tokens select KV groups, while each group retains all tokens' keys and values. (b) Queries and complete K/V histories are concatenated by group. (c) The mask enforces group assignments and causal order using original sequence positions. (d) Fully masked tiles are skipped; red entries show disallowed scores computed within the remaining tiles. These single-document examples use one query head per KV group. Panel (a) uses 12 tokens; panels (b–d) use 11. For multiple documents, the mask also prevents attention across document boundaries.

(KV4) and wider feedforward layers, matched in total parameter count to the sparse KV16/A4 model (16 total KV groups, four active per token). All twelve configurations are trained with two random seeds each. The larger family uses mixture-of-experts feedforward layers, with approximately 3B total parameters and 0.7B active parameters per token; we train four attention configurations with one seed each. All models are trained from scratch on 32K-token spans of ProLong (Gao et al., 2025), with document masking and the Llama 2 tokenizer's 32,000-token vocabulary. The 220M models are trained for 17,000 steps with 524,288 tokens per batch, for approximately 8.91B training tokens. The larger models are trained for 30,000 steps with 1,048,576 tokens per batch, for approximately 31.46B training tokens. Appendices C and D give architecture and training details for the two families.

4.2 Language Modeling Results

Table 1 reports results for the 220M family. WikiText (Merity et al., 2016) perplexity is normalized per whitespace-delimited word, and LAMBADA (Paperno et al., 2016) perplexity per example, scoring only the final target word. These different units, datasets, and prediction tasks make absolute perplexities incomparable across evaluations; we compare models within each metric.

We use LongPPL (Fang et al., 2024), which measures perplexity on tokens selected for their dependence on long context. This evaluates context use through next-token prediction, without requiring models to generate answers or follow task instructions.

Across the standard attention configurations, WikiText perplexities vary modestly, while LAMBADA shows larger differences. The dense KV4 model with wider feedforward layers achieves the lowest perplexity on both evaluations. Its LongPPL is nevertheless higher than that of the parameter-matched sparse KV16/A4 model (9.51 versus 9.03).

Table 1: Language-modeling results for the 220M family. Results use two training seeds (1234 and 1235) at step 17,000; lower is better. WikiText, LAMBADA, and LongPPL scores are arithmetic means across seeds. G and A are total and active KV-head counts; all configurations use eight active query heads. Underlined values are best within each A group; bold underlined values are best overall (per column, before rounding).

Model	G	A	WikiText PPL	LAMBADA PPL	LongPPL
Dense GQA	2	2	34.00	45.58	9.90
Sparse	8	2	33.81	45.62	9.42
Sparse	12	2	33.72	47.70	9.66
Sparse	16	2	33.80	47.88	9.87
Dense GQA	4	4	34.17	47.56	9.88
Dense GQA (wide FFN)	4	4	**33.05**	**41.09**	9.51
Sparse	8	4	33.73	46.13	9.33
Sparse	12	4	33.44	46.63	9.05
Sparse	16	4	33.34	43.95	9.03
Dense GQA	8	8	33.65	45.65	9.44
Sparse	12	8	33.42	42.72	8.89
Sparse	16	8	33.23	44.75	**8.85**

Table 2: Language-modeling results for the 3B feedforward-MoE family (approximately 0.7B active parameters per token). Each configuration uses one training seed (1234), evaluated at step 30,000, with 16 active query heads. G and A denote total and active KV heads. WikiText and LAMBADA use word and target-word PPL, respectively. LAMB. abbreviates LAMBADA. Other benchmarks are PIQA (Bisk et al., 2020), HellaSwag (Zellers et al., 2019), ARC-Easy/Challenge (ARC-E/C; Clark et al., 2018), SocialIQA (SIQA; Sap et al., 2019), and BoolQ (Clark et al., 2019). HellaSwag and ARC-C use normalized accuracy. Underlined values are best within each A group; bold underlined values are best overall (per column, before rounding).

			Accuracy (%) ↑							Perplexity ↓		↓
Model	G	A	LAMB.	PIQA	HellaSwag	ARC-E	ARC-C	SIQA	BoolQ	WikiText	LAMB.	LongPPL
Dense GQA	4	4	**51.19**	70.95	48.62	**60.90**	**28.75**	39.25	57.19	17.49	**10.05**	4.71
Sparse	12	4	50.67	71.33	48.64	59.55	27.65	39.05	**59.27**	17.42	10.16	4.59
Dense GQA	8	8	50.49	71.65	48.23	58.92	28.24	**39.66**	58.50	**17.40**	10.09	4.58
Dense GQA	16	16	50.07	**71.76**	**48.65**	59.97	28.07	**39.66**	54.01	17.45	10.38	**4.48**

Table 2 reports perplexities and benchmark accuracies for the four 3B feedforward-MoE models. These results use one seed per configuration. Sparse KV12/A4 has a similar LongPPL point estimate to dense KV8 (4.594 versus 4.577). The paired difference, sparse minus dense, is $+0.017$ (95% document-bootstrap CI: $[-0.081, 0.125]$). The interval contains zero, so this analysis does not resolve a difference between these checkpoints. The bootstrap procedure resamples the same 50 GovReport documents (Huang et al., 2021) and conditions on the trained checkpoints and fixed token selectors; it does not estimate training-seed variability. Appendix B gives the procedure and further paired comparisons.

4.3 Decode Speed and Aggregate Throughput

We distinguish *per-sequence decode speed* (TPS), the number of tokens generated per second for each sequence, from *aggregate decode throughput*, the total number generated per second across the batch. In our synchronous benchmarks, TPS is the reciprocal of decode-step time in seconds,

and aggregate throughput is batch size multiplied by TPS. *Peak aggregate throughput* is the highest observed aggregate throughput over the tested batch sizes for each model and context length.

Per-sequence decode speed. Table 3 reports whole-model, per-sequence decode speed for the 220M family at a common context length and batch size. Relative to dense GQA with the same active KV-group count, sparse models retain 83.0–87.1% of TPS for $A = 2$, 94.2–97.3% for $A = 4$, and 96.2–96.6% for $A = 8$.

Table 3: Whole-model decode speed for the 220M family at 64K context and batch size 64 on one NVIDIA B200. TPS denotes generated tokens per second per sequence, the reciprocal of decode-step time; higher is better. Timings use CUDA graphs, MXFP8 (E4M3) projections, and BF16 KV caches, with one benchmark result per configuration. Relative TPS uses the standard dense GQA baseline with the same active KV-group count A. G is the total KV-group count; all models use eight active query heads.

Model	G	A	TPS	Relative TPS (%)
Dense GQA	2	2	339.2	100.0
Sparse	8	2	295.3	87.1
Sparse	12	2	288.4	85.0
Sparse	16	2	281.4	83.0
Dense GQA	4	4	189.2	100.0
Dense GQA (wide FFN)	4	4	189.5	100.2
Sparse	8	4	183.6	97.0
Sparse	12	4	184.0	97.3
Sparse	16	4	178.2	94.2
Dense GQA	8	8	105.4	100.0
Sparse	12	8	101.9	96.6
Sparse	16	8	101.5	96.2

Peak aggregate throughput. Figure 1 (right) reports peak aggregate decode throughput for the 3B models at context lengths from 8K to 512K on one NVIDIA B200 using optimized BF16 CUDA graphs.

Sparse KV12/A4 achieves 1.60–1.94× the peak throughput of dense KV8, with a similar LongPPL point estimate (4.59 versus 4.58; Table 2). Relative to dense KV4, it retains 86–95% of peak throughput while achieving lower LongPPL (4.59 versus 4.71). Each model uses its own batch size at the observed throughput peak, in contrast to the fixed-batch TPS comparison in Table 3.

4.4 Attention Training Efficiency and Masked Computation

Masked computation within attention tiles. Figure 4 (left) reports the fraction of executed attention matrix-product FLOPs corresponding to logically masked interactions, estimated from valid query–key pairs and executed tile cells. For the three sparse configurations with two active KV groups, this fraction decreases from 22.8–35.0% at 2K tokens to 0.84–1.72% at 64K tokens. The dense baseline also computes masked entries within partially valid tiles; its masked fraction decreases from 11.1% to 0.39% over the same range.

These counts quantify the arithmetic overhead imposed by the chosen packing and tile granularity. For a fixed tiling that evaluates retained tiles using dense matrix products, masked entries within tiles containing required interactions are unavoidable work. Changing the packing, tile sizes, or kernel can change this overhead.

Training throughput relative to dense FlexAttention. Figure 4 (right) reports attention-operator forward-plus-backward throughput for the same configurations, normalized to each run's paired dense KV2 measurement. At 64K context, sparse models with 8, 12, and 16 total KV groups achieve 95.4%, 90.6%, and 86.6% of dense throughput, respectively.

Let f_s and f_d denote the sparse and dense masked-work fractions. For equal useful attention-product work, the ratio of executed FLOPs is $(1 - f_d)/(1 - f_s)$. For $G = 16$, $A = 2$ at 64K, this gives only 1.35% additional attention-product work, predicting 98.7% of dense throughput in a compute-only model with equal achieved arithmetic rates. This is a reference prediction, not a general lower bound on runtime slowdown: memory access, masking, scheduling, and hardware utilization can also affect performance. The measured 86.6% therefore cannot be explained by this work ratio alone. Because the tile-cell counts do not separately account for backward-kernel work, they do not provide an exact decomposition of the forward-plus-backward timing gap or identify its remaining causes.

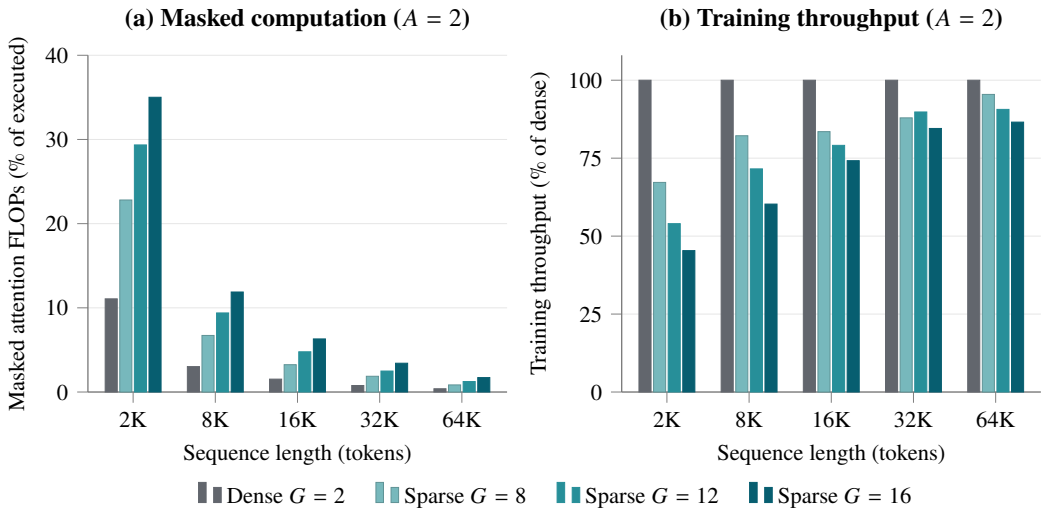

Figure 4: **Masked computation and sparse attention training throughput.** Dense KV2 and sparse configurations ($A = 2$, $G \in \{8, 12, 16\}$) use eight active query heads and full causal sequences with 131,072 tokens per batch. **Left:** Masked attention-product FLOPs as a percentage of executed FLOPs, $100(1 - N_{\text{valid}}/N_{\text{executed}})$, using valid query–key pairs and executed tile cells. **Right:** Attention-operator forward-plus-backward throughput on an NVIDIA B200, relative to each run's paired dense FlexAttention baseline (100%). These measurements concern the attention operator, not full-model training.

5 Related Work

Reducing KV storage. GQA (Ainslie et al., 2023), sliding-window attention, and recurrent–attention hybrids (Yang et al., 2024) reduce KV storage and traffic, but can impair long-context quality (Bertsch et al., 2026; Wang et al., 2025). We retain complete histories in a larger KV bank while reading only selected groups at each step.

Routing query heads. Mixture of Attention Heads (MoA) (Zhang et al., 2022) routes tokens to query/output experts sharing KV projections. Grouped Query Experts (GQE) (Tripathi & Kumar, 2026) routes among query heads within KV groups. Neither selects a subset of KV groups to reduce decode-time cache reads.

Routing attention heads and KV groups. Mixture-of-Head Attention (MoH) (Jin et al., 2024) studies token-dependent head routing, including training from scratch, but reports no KV-cached autoregressive decode benchmarks. LLaMA-MoE v2 (Qu et al., 2024) forms attention experts through post-training. Its released implementation also routes KV writes, so each group stores only selected tokens.[1] Our groups retain every token's keys and values.

[1] See the expert-specific projections and cache updates in the authors' attention implementation.

Selective KV access for decoding. Polar Sparsity (Shrestha et al., 2025), the closest prior work, trains head predictors on a frozen pretrained dense model and accelerates batched decoding through selective KV access while preserving complete histories. We jointly train the KV representations and routing from scratch to study quality as total KV capacity increases at fixed active-group counts. Our parallel implementation enables this training without evaluating every group's attention for every token.

6 DISCUSSION

Our results show that sparse KV routing can improve the long-context quality of GQA models while retaining most of their decode advantage. At 220M scale, sparse KV12/A4 improves LongPPL from dense KV4's 9.88 to 9.05 while retaining 97.3% of per-sequence decode speed at 64K context and batch size 64. At 3B scale, sparse KV12/A4 improves on dense KV4's LongPPL while retaining 86–95% of its peak aggregate throughput. It also approaches dense KV8's LongPPL (4.59 versus 4.58) with 1.60–$1.94\times$ the peak aggregate throughput. These comparisons demonstrate the benefit of learning a larger set of KV representations while accessing only a few at each step.

The additional capacity requires more cache storage. Every KV group retains the complete sequence history so that future tokens can select any group, and the larger cache reduces the batch size that fits in GPU memory. The 3B peak-throughput measurements include this cost: each model is measured at its own best observed batch size. Despite this storage cost, the sparse model preserves most of dense KV4's throughput and exceeds dense KV8's throughput at every tested context length. Thus, the storage-for-bandwidth tradeoff yields a better measured quality–throughput balance in these comparisons.

The parallel training results establish that this design can be trained without evaluating attention for every group and discarding unselected outputs. At 64K context, the tested sparse attention operators with two active KV groups reach 86.6–95.4% of matched dense forward-plus-backward throughput. Only 0.84–1.72% of their executed attention-product work is masked. The compute-only comparison in Section 4.4 predicts a much smaller penalty than observed, motivating further work on implementation performance beyond reducing computation on masked entries.

7 CONCLUSION

Mixture of KV Groups separates total KV capacity from the KV groups accessed by each token. Models trained from scratch with this architecture improve long-context quality over GQA with the same active KV-group count while retaining most of its decode speed and aggregate throughput in the evaluated configurations. The sparse 3B model delivers substantially higher peak aggregate throughput than dense KV8 with similar measured LongPPL. Our FlexAttention implementation and measurements also show that sparse KV routing supports parallel training with modest attention-operator overhead at long contexts.

AI USE STATEMENT

Generative AI tools were used to help refine hypotheses and experimental comparisons, survey and summarize related literature, suggest the manuscript structure and title, draft and edit portions of the manuscript, and format references. The authors reviewed the cited sources and all AI-assisted text and remain responsible for the final content, claims, and artifacts in this work.

ETHICS STATEMENT

This work studies computational efficiency in language models and does not involve human subjects. The proposed method may reduce the compute required for long-context inference relative to dense attention with the same total head bank, but it does not by itself address the social risks of the language models in which it is used.

Reproducibility statement

The main paper defines the method and cache semantics and reports the evaluation and performance-measurement settings. The appendices provide the sparse attention implementation, model architectures and training hyperparameters, and the paired-bootstrap procedure for LongPPL confidence intervals.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

Amanda Bertsch, Luca Soldaini, Matthew R. Gormley, Graham Neubig, Hannaneh Hajishirzi, Kyle Lo, and Dirk Groeneveld. Cracks in the foundation: Seemingly minor architectural choices impact long context extension, 2026. URL https://arxiv.org/abs/2608.10296.

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.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2924–2936, 2019. doi: 10.18653/v1/N19-1300. URL https://aclanthology.org/N19-1300/.

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.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022. URL https://arxiv.org/abs/2205.14135.

Lizhe Fang, Yifei Wang, Zhaoyang Liu, Chenheng Zhang, Stefanie Jegelka, Jinyang Gao, Bolin Ding, and Yisen Wang. What is wrong with perplexity for long-context language modeling?, 2024. URL https://arxiv.org/abs/2410.23771.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL https://jmlr.org/papers/v23/21-0998.html.

Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. How to train long-context language models (effectively). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. URL https://arxiv.org/abs/2410.02660.

Driss Guessous, Yanbo Liang, Joy Dong, and Horace He. FlexAttention: The flexibility of PyTorch with the performance of FlashAttention. PyTorch Blog, 2024. URL https://pytorch.org/blog/flexattention/.

Luyang Huang, Shuyang Cao, Nikolaus Parulian, Heng Ji, and Lu Wang. Efficient attentions for long document summarization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1419–1436, 2021. doi: 10.18653/v1/2021.naacl-main.112. URL https://aclanthology.org/2021.naacl-main.112/.

Peng Jin, Bo Zhu, Li Yuan, and Shuicheng Yan. MoH: Multi-head attention as mixture-of-head attention, 2024. URL https://arxiv.org/abs/2410.11842.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. URL https://arxiv.org/abs/1609.07843.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, 2016. doi: 10.18653/v1/P16-1144. URL https://aclanthology.org/P16-1144/.

Xiaoye Qu, Daize Dong, Xuyang Hu, Tong Zhu, Weigao Sun, and Yu Cheng. LLaMA-MoE v2: Exploring sparsity of LLaMA from perspective of mixture-of-experts with post-training, 2024. URL https://arxiv.org/abs/2411.15708.

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 (EMNLP-IJCNLP)*, pp. 4463–4473, 2019. doi: 10.18653/v1/D19-1454. URL https://aclanthology.org/D19-1454/.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations*, 2017. URL https://arxiv.org/abs/1701.06538.

Susav Shrestha, Brad Settlemyer, Nikoli Dryden, and Narasimha Reddy. Polar sparsity: High throughput batched LLM inferencing with scalable contextual sparsity. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2505.14884.

Vishesh Tripathi and Abhay Kumar. Grouped query experts: Mixture-of-experts on GQA self-attention, 2026. URL https://arxiv.org/abs/2606.20945.

Dustin Wang, Rui-Jie Zhu, Steven Abreu, Yong Shan, Taylor Kergan, Yuqi Pan, Yuhong Chou, Zheng Li, Jibin Wu, Ge Zhang, Wenhao Huang, and Jason Eshraghian. A systematic analysis of hybrid linear attention, 2025. URL https://arxiv.org/abs/2507.06457.

Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. Auxiliary-loss-free load balancing strategy for mixture-of-experts, 2024. URL https://arxiv.org/abs/2408.15664.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule, 2024. URL https://arxiv.org/abs/2412.06464.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026. URL https://arxiv.org/abs/2603.05451.

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, 2019. doi: 10.18653/v1/P19-1472. URL https://aclanthology.org/P19-1472/.

Xiaofeng Zhang, Yikang Shen, Zeyu Huang, Jie Zhou, Wenge Rong, and Zhang Xiong. Mixture of attention heads: Selecting attention heads per token. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 4150–4162, 2022. doi: 10.18653/v1/2022.emnlp-main.278.

A Parallel Sparse Attention Implementation

Listing 1 implements the attention core from Section 3.2, given projected Q/K/V and selected KV-group IDs. In the code, `L` is sequence length n, `G` is the total KV-group count G, `R` is the number of query heads per group H/G, and `D` is head dimension d_h. The caller supplies Q/K normalization, any positional transforms, and routing assignments; router weighting and output projections are outside this function. Document starts must describe the actual contiguous document partition, and each token's selected group IDs must be distinct. The explicit `FLASH` backend requires a compatible PyTorch, FlashAttention-4 (Zadouri et al., 2026), and CUDA environment; the default mask-block sizes are $(256, 128)$.

Listing 1: Packed sparse attention with causal and document masking.

```python
import torch
from torch.nn.attention.flex_attention import BlockMask, flex_attention


def _compact(active):
    width = active.shape[-1]
    destination = torch.where(active, active.cumsum(-1) - 1, width)
    index = torch.arange(width, device=active.device, dtype=torch.int32)
    source = torch.where(active, index, 0)
    return torch.zeros(
        *active.shape[:-1], width + 1,
        device=active.device, dtype=torch.int32,
    ).scatter_add(-1, destination, source)[..., :-1]


def _block_mask(lower, upper, kv_length, block_size):
    batch, rows = lower.shape
    q_tile, kv_tile = block_size
    q_blocks, kv_blocks = rows // q_tile, kv_length // kv_tile
    width = kv_blocks + 1
    offset = (
        torch.arange(batch * rows, device=lower.device) // q_tile * width
    )
    begin = offset + lower.flatten().long() // kv_tile
    end = offset + upper.flatten().long() // kv_tile + 1
    difference = torch.zeros(
        batch * q_blocks * width, device=lower.device, dtype=torch.int32
    )
    ones = torch.ones_like(begin, dtype=torch.int32)
    difference.scatter_add_(0, begin, ones)
    difference.scatter_add_(0, end, -ones)
    active = (
        difference.view(batch, q_blocks, width)[..., :-1].cumsum(-1) > 0
    )
    left = lower.view(batch, q_blocks, q_tile).amax(-1)
    right = upper.view(batch, q_blocks, q_tile).amin(-1)
    start = torch.arange(kv_blocks, device=lower.device) * kv_tile
    full = (start >= left[..., None]) & (
        start + kv_tile - 1 <= right[..., None]
    )
    partial = active & ~full

    def mask_mod(b, _head, q, k):
        return (k >= lower[b, q]) & (k <= upper[b, q])

    return BlockMask.from_kv_blocks(
        partial.sum(-1).int()[:, None], _compact(partial)[:, None],
        full.sum(-1).int()[:, None], _compact(full)[:, None],
        BLOCK_SIZE=block_size, mask_mod=mask_mod,
        seq_lengths=(rows, kv_length),
    )
```

```python
_compiled_mask = torch.compile(_block_mask, fullgraph=True)
_compiled_attention = torch.compile(flex_attention, fullgraph=True)


def sparse_flex_attention(
    q, k, v, heads, document_starts=None, *, block_size=(256, 128)
):
    """Q[B,L,A,R,D], K/V[B,G,L,D], selected KV IDs heads[B,L,A].

    R is the number of query heads per KV group. Each Q selection slot
    corresponds to heads; IDs are distinct within each token's selection.
    document_starts[B,L] specifies a contiguous document partition, or
    defaults to one document per sequence. Caller applies Q/K normalization
    and any positional transforms. Returns outputs in Q's
    token/selection/head order,
    before output projection and router weighting.
    Default mask blocks target B200 with the FlashAttention-4 backend.
    """
    if q.ndim != 5 or k.ndim != 4 or v.shape != k.shape:
        raise ValueError("Expected Q[B,L,A,R,D] and K/V[B,G,L,D]")
    batch, length, active, queries_per_group, dim = q.shape
    total = k.shape[1]
    if (
        min(*q.shape, total) <= 0 or active > total
        or k.shape != (batch, total, length, dim)
        or heads.shape != (batch, length, active)
        or total * length >= 2**31
    ):
        raise ValueError("Incompatible dimensions or head assignments")
    if (
        any(t.device != q.device for t in (k, v, heads))
        or not q.dtype == k.dtype == v.dtype
    ):
        raise ValueError(
            "Q/K/V must share dtype; inputs must share device"
        )
    if heads.dtype not in (torch.int32, torch.int64):
        raise ValueError("Head assignments must be integer tensors")
    if document_starts is None:
        document_starts = torch.zeros(
            batch, length, device=q.device, dtype=torch.int32
        )
    if (
        document_starts.shape != (batch, length)
        or document_starts.device != q.device
        or document_starts.dtype not in (torch.int32, torch.int64)
    ):
        raise ValueError(
            "Expected integer document_starts[B,L] on Q's device"
        )
    if (
        min(block_size) <= 0 or length % block_size[1]
        or length * active * queries_per_group % block_size[0]
    ):
        raise ValueError("Sequence and packed-query lengths must align")
    positions = torch.arange(length, device=q.device)
    torch._assert_async(
        ((heads >= 0) & (heads < total)).all(), "Invalid KV head ID"
    )
    torch._assert_async(
        ((document_starts >= 0) & (document_starts <= positions)).all(),
        "Invalid document start",
    )
    assignments = heads.flatten(1)
    order = assignments.argsort(dim=1, stable=True)
    index = order[:, :, None, None].expand(-1, -1, queries_per_group, dim)
    packed_q = q.flatten(1, 2).gather(1, index).reshape(batch, 1, -1, dim)
```

```
    position = order // active
    head_offset = assignments.gather(1, order) * length
    lower = (head_offset + document_starts.gather(1, position))
    lower = lower.repeat_interleave(queries_per_group, 1).int()
    upper = (head_offset + position)
    upper = upper.repeat_interleave(queries_per_group, 1).int()
    mask = _compiled_mask(lower, upper, total * length, block_size)
    output = _compiled_attention(
        packed_q, k.reshape(batch, 1, total * length, dim),
        v.reshape(batch, 1, total * length, dim),
        block_mask=mask, kernel_options={"BACKEND": "FLASH"},
    ).view(batch, length * active, queries_per_group, dim)
    return torch.empty_like(output).scatter(1, index, output).view_as(q)
```

B PAIRED LONGPPL CONFIDENCE INTERVALS

We resample 50 GovReport documents (Huang et al., 2021) with replacement, using the same sampled indices across models, checkpoints, and the three fixed token selectors: Meta-Llama-3.1-8B, Mistral-Large-Instruct-2407, and Qwen2-72B-Instruct. Within each replicate and checkpoint, we sum selected-token negative log-likelihoods and token counts separately for each selector, divide to obtain its mean cross-entropy, and exponentiate the average of the three selector cross-entropies. This yields the geometric mean of the three selector perplexities. For 220M models, we then average the scores of the two fixed training checkpoints; each 3B model has one checkpoint.

We use 50,000 replicates for 220M and 20,000 for 3B, with bootstrap seeds 20260925 and 1234, respectively. Table 4 reports the 2.5th and 97.5th percentiles of the paired differences. These nominal 95% intervals are not adjusted for multiple comparisons. They measure evaluation-document uncertainty conditional on the available checkpoints and fixed selectors, not training-seed variability or transfer to another corpus. An interval containing zero does not establish equivalence.

Table 4: Paired LongPPL differences and 95% document-bootstrap intervals. Differences are candidate minus baseline; negative values favor the candidate. Wide FFN denotes dense KV4 widened to match the total parameter count of sparse KV16/A4, not the other sparse configurations.

Candidate	Baseline	Difference	95% CI
220M: same active KV-group count			
Sparse KV8/A2	Dense KV2	-0.484	$[-0.824, -0.230]$
Sparse KV12/A2	Dense KV2	-0.244	$[-0.487, -0.003]$
Sparse KV16/A2	Dense KV2	-0.037	$[-0.303, 0.265]$
Sparse KV8/A4	Dense KV4	-0.556	$[-0.888, -0.310]$
Sparse KV12/A4	Dense KV4	-0.838	$[-1.250, -0.552]$
Sparse KV16/A4	Dense KV4	-0.850	$[-1.246, -0.586]$
Sparse KV12/A8	Dense KV8	-0.553	$[-0.830, -0.326]$
Sparse KV16/A8	Dense KV8	-0.590	$[-0.915, -0.319]$
220M: wider-feedforward control			
Sparse KV8/A4	Dense KV4 (wide FFN)	-0.181	$[-0.400, 0.011]$
Sparse KV12/A4	Dense KV4 (wide FFN)	-0.463	$[-0.784, -0.227]$
Sparse KV16/A4	Dense KV4 (wide FFN)	-0.475	$[-0.755, -0.279]$
3B			
Dense KV8	Dense KV4	-0.134	$[-0.290, -0.020]$
Dense KV16	Dense KV4	-0.231	$[-0.411, -0.101]$
Sparse KV12/A4	Dense KV4	-0.117	$[-0.248, -0.013]$
Dense KV16	Dense KV8	-0.097	$[-0.195, -0.003]$
Sparse KV12/A4	Dense KV8	$+0.017$	$[-0.081, 0.125]$
Sparse KV12/A4	Dense KV16	$+0.114$	$[0.022, 0.228]$

C 220M Model Architecture and Training

Shared architecture. The 220M family consists of decoder-only hybrids with model dimension $d = 1{,}024$ and eight sequence-mixing blocks in the order GDN, GDN, attention, GDN, attention, GDN, attention, GDN. Each sequence mixer is followed by a dense SwiGLU feedforward sublayer; both sublayers have residual connections and use RMS normalization in their residual branches. The untied input embedding and language-model head use a 32,000-token vocabulary and together contain 65,536,000 parameters. Full-attention layers use per-head Q/K RMSNorm and head dimension $d_h = 128$. These hybrids use neither positional embeddings nor RoPE.

Each GDN layer has six key/value heads of width 128, value expansion one, and total key and value widths of 768. It uses a gated output, a causal short convolution of width four, and a chunk-parallel training kernel. The standard feedforward sublayers use Transformer Engine's fused RMSNorm–SwiGLU MLP with intermediate width $h = 4{,}864$, no biases, and no dropout.

Attention configurations. Every configuration activates eight query heads per token. Dense models have eight total query heads and $G = A \in \{2, 4, 8\}$ KV groups. Sparse models have $G \in \{8, 12, 16\}$ total KV groups and activate A of them, with $8/A$ query heads per group and $H = 8G/A$ total query heads. Table 5 lists the evaluated configurations and exact trainable parameter counts, including embeddings and the output head. Subtracting 65,536,000 gives the count excluding those two matrices. The parameter-matched dense KV4 control changes the eight feedforward widths, in layer order, to $[6144, 5888, 6144, 5888, 6144, 5888, 5888, 6144]$. Its total count is 49,152 parameters (0.020%) below sparse KV16/A4.

Table 5: Architecture configurations for the 220M family. G and A count total and active KV groups, H counts total query heads, and H/G is the number of query heads per group. All configurations activate eight query heads per token and use seeds 1234 and 1235.

Model	G	A	H	H/G	Total parameters
Dense GQA	2	2	8	4	212,740,540
Sparse	8	2	32	4	236,358,076
Sparse	12	2	48	4	252,099,004
Sparse	16	2	64	4	267,839,932
Dense GQA	4	4	8	2	214,313,404
Dense GQA (wide FFN)	4	4	8	2	242,624,956
Sparse	8	4	16	2	223,775,164
Sparse	12	4	24	2	233,224,636
Sparse	16	4	32	2	242,674,108
Dense GQA	8	8	8	1	217,459,132
Sparse	12	8	12	1	223,787,452
Sparse	16	8	16	1	230,091,196

Routing and attention implementations. Each sparse layer has an independent bias-free linear router. FP32 sigmoid scores weight the selected group outputs without renormalization; selection adds a non-gradient group bias before top-A selection. After each optimizer step, routed-token counts are summed across data-parallel ranks and the biases are updated using Equation 6 with $\eta = 0.001$. There is no auxiliary routing loss. Sparse query and output projections use Transformer Engine grouped linear layers; K/V projections are dense over all groups. Compiled FlexAttention enforces group isolation, causal order, and document-segment boundaries. Dense attention uses variable-length causal FlashAttention-4 (Zadouri et al., 2026). Both variants use Transformer Engine projections and per-head Q/K RMSNorm.

C.1 PRETRAINING CONFIGURATION

Table 6: Shared pretraining settings for all 220M configurations.

Setting	Value
Data and tokenizer	ProLong 50B corpus retokenized with Llama 2; vocabulary 32,000, EOS ID 2
Optimizer	Fused AdamW; $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$
Weight decay / clipping	0.1 / global gradient norm 1.0
Peak / final learning rate	4×10^{-4} / 4×10^{-5}
Learning-rate schedule	1,000-step linear warmup, 14,000 stable steps, 2,000-step linear cooldown
Training steps	17,000
Tokens per optimizer step	524,288
Total input tokens	8,912,896,000
Hardware and parallelism	Eight NVIDIA B200 GPUs; one DDP rank per GPU
Per-rank input buffer	65,536 tokens; gradient accumulation 1
Maximum logical segment	32,768 tokens
Configured sequence capacity	131,072 tokens
Activation checkpointing	Disabled for attention, GDN, and FFN layers
Full-model compilation	Disabled; sparse FlexAttention is compiled
Checkpoint interval	Every 1,000 steps
Validation	Every 500 steps; 64 examples, maximum context 32,768
Seeds	1234 and 1235

Packing and objective. Each data-parallel rank reads 65,536 tokens per step from the packed stream. EOS tokens define document boundaries. Documents longer than 32,768 tokens are split into segments of at most that length without adding synthetic EOS tokens. Attention masks and GDN state resets prevent information flow across document or segment boundaries, and next-token targets crossing a boundary are excluded from the loss. All configurations and seeds traverse the same deterministic token order; seeds change initialization and stochastic state, not data order. The configured 131,072-token capacity is distinct from the maximum logical training context of 32,768 tokens. The objective is next-token cross-entropy, evaluated using a fused linear-cross-entropy loss.

Numerical precision. Training uses BF16 autocast. Transformer Engine linear and MLP operations, including dense attention projections and sparse grouped query/output projections, use MXFP8 block scaling with E4M3 values. GDN operations use BF16. Router logits, sigmoid scores, selection biases, and routing decisions use FP32, as do model parameters and optimizer states. No dropout or auxiliary routing loss is used.

D 3B MODEL ARCHITECTURE AND TRAINING

Shared architecture. The larger family consists of decoder-only GDN–attention hybrids with model dimension $d = 2{,}048$ and 16 sequence-mixing blocks. Full-context attention appears at blocks 4, 6, 8, 10, 12, and 14 (one-based indexing); the other ten blocks use GDN. Every mixer is followed by a feedforward sublayer. Mixer and feedforward outputs have residual connections, with RMSNorm in the residual branches. Full attention uses per-head Q/K RMSNorm, head dimension $d_h = 128$, and no positional embedding or RoPE. The untied 32,000-token input embedding and output head contain 131,072,000 parameters. Each GDN layer has eight key/value heads of width 128, value expansion one, and key and value widths of 1,024. It uses a gated output, a causal short convolution of width four, and a chunk-parallel training kernel.

Feedforward layers. The first feedforward sublayer is a dense Transformer Engine SwiGLU MLP with intermediate width $h = 8{,}192$. The remaining 15 are SwiGLU MoE layers, each with 24 experts, four active experts per token, and intermediate width $h = 1{,}024$ per expert. There are no shared experts or dropped tokens. Expert gate/up projections are fused; both expert projections use Transformer Engine grouped linear operations. The expert matrices contain 2,264,924,160 parameters in total, of which 377,487,360 are selected per token. These counts concern expert matrices, not total model FLOPs.

Attention configurations. All configurations activate 16 query heads per token. Dense models have $H = 16$ total query heads and $G = A \in \{4, 8, 16\}$ KV groups. The sparse model has $G = 12$, $A = 4$, and four query heads per group, giving $H = 48$ total query heads. Table 7 gives exact trainable parameter counts; subtracting 131,072,000 gives counts excluding the embedding and output head. The "3B" label denotes the scale class: the checkpoints contain 2.615B–2.741B trainable parameters. These specifications and counts are verified from the saved step-30,000 checkpoint metadata and state dictionaries.

Table 7: Architecture configurations for the 3B family. G and A count total and active KV groups; H counts total query heads. All configurations activate 16 query heads per token and use seed 1234.

Model	G	A	H	H/G	Total parameters
Dense GQA	4	4	16	4	2,615,421,856
Sparse	12	4	48	4	2,741,398,432
Dense GQA	8	8	16	2	2,628,004,768
Dense GQA	16	16	16	1	2,653,170,592

Feedforward and attention routing. Both routers are bias-free FP32 linear projections, with sigmoid scores and top-four selection after adding non-gradient selection biases. Feedforward MoE normalizes the selected unbiased scores over the selected experts before weighting their outputs. The separate attention router uses the raw selected sigmoid scores without renormalization. For both routers, counts are summed over data-parallel ranks after each optimizer step, and the selection biases follow Equation 6 with $\eta = 0.001$. Biases are checkpointed buffers, not AdamW parameters; both auxiliary-loss coefficients are zero.

Sparse query/output projections use grouped linear layers; K/V projections are dense over all 12 groups. Compiled FlexAttention packs queries by KV group and enforces causal order, group isolation, and packed-segment boundaries. Each selected group exposes its complete preceding KV history within the segment, regardless of earlier routing decisions. Dense training attention uses variable-length causal FlashAttention-4 (Zadouri et al., 2026). Both projection paths use Transformer Engine FP8 operations.

D.1 PRETRAINING CONFIGURATION

Table 8: Shared pretraining settings for all four 3B configurations.

Setting	Value
Data and tokenizer	ProLong 50B corpus retokenized with Llama 2; vocabulary 32,000, EOS ID 2
Objective	Next-token cross-entropy
Optimizer	Fused AdamW; $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$
Weight decay / clipping	0.1 / global gradient norm 1.0
Peak / final learning rate	4×10^{-4} / 4×10^{-5}
Learning-rate schedule	1,000-step linear warmup, 25,000 stable steps, 4,000-step linear cooldown
Training steps	30,000
Tokens per optimizer step	1,048,576
Total input tokens	31,457,280,000
Hardware and parallelism	Eight NVIDIA B200 GPUs; one replicated DDP rank per GPU
Per-rank microstep buffer	65,536 tokens
Gradient accumulation	Two microsteps
Maximum logical segment	32,768 tokens
Configured sequence capacity	131,072 tokens
Activation checkpointing	Disabled for attention, GDN, and FFN layers
Full-model compilation	Disabled; sparse FlexAttention is compiled
Checkpoint interval	Every 1,000 steps
Validation	Every 500 steps; 64 examples, maximum context 32,768
Seed	1234

Packing and boundaries. Each rank reads 65,536-token physical buffers from a continuous token stream. Eight ranks and two gradient-accumulation microsteps yield 1,048,576 tokens per optimizer step. EOS tokens define document boundaries. The `fixed_blocks_v1` protocol also inserts boundaries at globally aligned 32,768-token intervals, so no logical segment exceeds 32K tokens. Attention masks and GDN state resets prevent information flow across either boundary type, and next-token targets crossing segment boundaries are excluded from the loss. The configured 131,072-token capacity is distinct from the maximum logical training context of 32,768 tokens.

Numerical precision. Training uses BF16 autocast. Transformer Engine linear, dense MLP, and MoE grouped-linear operations use MXFP8 block scaling with E4M3 values; GDN operations use BF16. Attention and MoE router logits, sigmoid scores, selection biases, and routing decisions use FP32. Model parameters and optimizer states are maintained in FP32. No dropout, activation checkpointing, or auxiliary routing loss is used.