BATCH MULTIPLEXING LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Autoregressive decoding repeatedly applies the same model to many independent sequences. We study batch multiplexing, a small change to a decoder-only transformer that lets several sequences share part of an attention computation while still producing one next-token distribution per sequence. In selected middle layers, a learned multiplexer combines K same-position hidden states into one attention input. An ordinary attention block runs once per group, and a demultiplexer returns its update to the original sequences before each sequence runs its own MLP. We compare a family of depth-scaled batch-multiplexed models against nanochat using strictly controlled protocols. Across model depths and training horizons, batch multiplexing improves the validation-quality/decode-throughput frontier, which suggests that it is a promising direction for further exploration.

1 INTRODUCTION

Autoregressive decoding repeatedly applies the same model to one new token from each active request. Batching improves hardware utilization, but it does not reduce the attention computation assigned to an individual request: every sequence still produces its own query, reads its own growing KV cache, and receives its own attention output. As a result, attention cost continues to grow with the number of independent sequences in a serving batch.

Recent attention architectures reduce this cost by exploiting redundancy along several dimensions. Multi-query and grouped-query attention share key-value heads (Shazeer, 2019; Ainslie et al., 2023); multi-head latent attention compresses their representation (DeepSeek-AI, 2024). For long contexts, sparse-attention methods restrict each query to a subset of the history, using either structured patterns or content-dependent selection (Child et al., 2019; Roy et al., 2021). Recent systems such as DeepSeek Sparse Attention implement this selection with a learned indexer (DeepSeek-AI, 2025), while Compressed Sparse Attention and Heavily Compressed Attention aggregate KV entries along the sequence dimension before attention (DeepSeek-AI, 2026). These methods leave one source of redundancy largely untouched: even when independent requests are decoded together, attention is still computed separately for each sequence. This raises a different question: can a model learn to share attention across independent sequences while preserving a separate next-token prediction for each one?

We study this question with *batch multiplexing*. As illustrated in Figure 1, the beginning and end of the network process each sequence independently. In selected middle blocks, a learned multiplexer combines the same-position hidden states of K sequences into one attention input. An ordinary causal attention module runs once for the group, and a demultiplexer converts the shared attention output into a separate update for each sequence. Each sequence then continues with its own residual state and MLP computation. Thus only attention and its corresponding KV state are shared; the model still produces one next-token distribution per sequence. This operation compresses a different axis from existing efficient-attention methods.

To determine whether this architectural change improves the quality–throughput trade-off, we compare batch-multiplexed model families with nanochat baselines across depths and two training horizons (Karpathy, 2025). At each depth, both models have the same backbone shape, follow the same nanochat training recipe, and are trained on the same number of raw tokens. We measure validation BPB, CORE, and decode throughput under a common evaluation protocol.

Across depths and training horizons, batch multiplexing improves the empirical validation-quality/decode-throughput frontier. Slot-count experiments measure how quality changes as more

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.

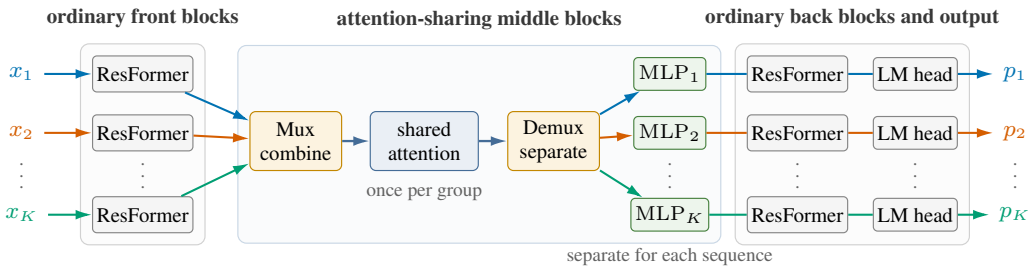


Figure 1: Batch multiplexing for one group of K independent sequences. Ordinary nanochat blocks process each sequence separately at the beginning and end of the network. In the middle blocks, a multiplexer forms one shared attention input, and a demultiplexer returns the attention update to every sequence.

sequences share attention. Write-weight analyses further show that the learned multiplexer assigns systematically different importance to tokens within a group, and that the resulting ranking closely tracks the next-token difficulty measured by the baseline model.

Our main contributions are:

- We introduce batch multiplexing for autoregressive decoders, sharing selected attention computations across independent sequences while preserving per-sequence residual states, MLP computation, and next-token distributions.
- We compare nanochat and batch-multiplexed model families across depth under matched training settings and evaluate their serving efficiency with measured decode throughput.
- We show improved quality–throughput frontiers at both training horizons and use slot-count and write-weight analyses to study how group size affects quality and how the multiplexer distributes influence among sequences.

2 Batch Multiplexing

2.1 Architecture

Batch multiplexing reorganizes a batch of B independent sequences into $G = B/K$ groups, each with K slots. Within each group, the sequences pass through ordinary nanochat blocks (Karpathy, 2025) at the beginning and end of the network. In the middle blocks, their states at the same position are combined into one attention input. Attention is computed once for the group in each middle block, and its update is returned to each sequence before that sequence's own MLP. Different groups do not interact; for shared attention, they are ordinary batch elements.

Let $X \in \mathbb{N}^{G\times K\times T}$ be a batch of G groups, each holding K token sequences of length T. Let E be the token embedding table, and let f_{front} be the ordinary nanochat blocks before the middle region. These blocks use the same parameters for every sequence. For slot s, they produce a separate hidden state

$$e_s = E(X_{:,s,:}), \qquad u_s = f_{\text{front}}(e_s), \qquad u_s \in \mathbb{R}^{G\times T\times d}. \tag{1}$$

At a middle block ℓ, let $x_{s,\ell}$ be the incoming state of sequence s, with $x_{s,0} = u_s$. The block applies nanochat's residual mixing to each sequence separately. It then combines the resulting states into one attention input per group:

$$r_{s,\ell} = \alpha_\ell x_{s,\ell} + \beta_\ell x_{s,0}, \qquad h_\ell = \mathrm{Mux}_\ell(r_{1,\ell}, \ldots, r_{K,\ell}), \qquad h_\ell \in \mathbb{R}^{G\times T\times d}. \tag{2}$$

With $n(\cdot)$ denoting RMSNorm and QKNorm denoting query-key normalization (Zhang & Sennrich, 2019; Henry et al., 2020), the same nanochat attention module now runs on the G shared rows. For a normalized input u and value-embedding signal a, its computation is

$$\begin{aligned} q = W_\ell^q u, \quad k = W_\ell^k u, \quad v = W_\ell^v u, \\ \mathrm{Attention}_\ell(u, a) = W_\ell^o\, \mathrm{Attn}(\mathrm{QKNorm}(q), \mathrm{QKNorm}(k), v + a)\,. \end{aligned} \tag{3}$$

Applying it once per group gives

$$z_\ell = \mathrm{Attention}_\ell\big(n(h_\ell), a_\ell^{\mathrm{shared}}\big), \qquad z_\ell \in \mathbb{R}^{G\times T\times d}. \tag{4}$$

The demultiplexer assigns this attention update back to each sequence. Each sequence then completes the block with its own nanochat MLP:

$$\begin{aligned} d_{s,\ell} &= \mathrm{Demux}_\ell(r_{s,\ell}, z_\ell), \\ y_{s,\ell} &= r_{s,\ell} + d_{s,\ell}, \\ x_{s,\ell+1} &= y_{s,\ell} + \mathrm{MLP}_\ell(n(y_{s,\ell})). \end{aligned} \tag{5}$$

Each middle block repeats this computation, passing a complete per-sequence state to the next block. Let ℓ_{last} be the final middle block, and let f_{back} denote the ordinary nanochat blocks after it. The same back stack is applied separately to every slot:

$$v_s = f_{\mathrm{back}}(x_{s,\ell_{\mathrm{last}}+1}), \qquad v_s \in \mathbb{R}^{G\times T\times d}. \tag{6}$$

Let H_s be sequence s's state after nanochat's final processing. The same untied LM head then scores every sequence:

$$\ell_{s,t} = W_{\mathrm{out}} H_{s,t}, \qquad p_{s,t} = \mathrm{softmax}(\mathrm{softcap}(\ell_{s,t})). \tag{7}$$

To compress the K sequence states in a group into a single attention input, the multiplexer partitions the hidden dimension into J channel ranges. Each range has its own set of sequence weights, allowing different parts of the shared representation to emphasize different sequences. Let $[v]_j$ denote the slice of a vector v corresponding to channel range j. For each range, the multiplexer computes one score for every sequence, normalizes these scores across the K sequences, and uses the resulting weights to form a weighted sum:

$$\begin{aligned} \omega_{s,\ell,j} &= \mathrm{softmax}_s\big([W_\ell^{\mathrm{score}} n(r_{s,\ell})]_j\big), \\ h_{\ell,j} &= \sum_{s=1}^{K} \omega_{s,\ell,j} [W_\ell^{\mathrm{write}} r_{s,\ell}]_j, \\ h_\ell &= \mathrm{Concat}_{j=1}^{J}(h_{\ell,j}). \end{aligned} \tag{8}$$

Concatenating the J weighted sums produces the shared attention input h_ℓ.

The demultiplexer reads the resulting attention *update* z_ℓ and assigns it back to the original sequences. For each channel range it forms a query from the original sequence state and a key from the shared update:

$$\begin{aligned} q_{s,\ell,j} &= [W_\ell^Q n(r_{s,\ell})]_j, \\ k_{\ell,j} &= [W_\ell^K n(z_\ell)]_j, \\ \pi_{s,\ell,j} &= \mathrm{softmax}_s\left(q_{s,\ell,j}^{\mathsf{T}} k_{\ell,j} / \sqrt{d_q}\right), \\ d_{s,\ell,j} &= \pi_{s,\ell,j} [W_\ell^R z_\ell]_j. \end{aligned} \tag{9}$$

After concatenation, $d_{s,\ell}$ is added to sequence s as in Equation 5; its MLP still runs on that sequence alone.

Value embeddings modify the value vectors inside nanochat attention. Nanochat uses a ResFormer-style variant of value residual learning (Zhou et al., 2024). For selected layers $\ell \in \mathcal{V}$, nanochat has a token-indexed table V_ℓ and a learned gate g_ℓ. An ordinary layer uses the original sequence's token and attention input. With the slot index restored, this is

$$a_{\ell,s} = g_\ell(n(r_{s,\ell})) \odot V_\ell(X_{:,s,:}). \tag{10}$$

In a middle layer, the attention input is shared. Its value embedding also combines the original sequences' token-indexed embeddings:

$$\begin{aligned} V_\ell^{\mathrm{shared}} &= \frac{\sum_{s=1}^{K} m_s V_\ell(X_{:,s,:})}{\sqrt{\max(1, \sum_{s=1}^{K} m_s)}}, \\ a_\ell^{\mathrm{shared}} &= g_\ell(n(h_\ell)) \odot V_\ell^{\mathrm{shared}}. \end{aligned} \tag{11}$$

Here m_s indicates whether slot s is active at that position. Layers without value embeddings use $a_\ell = 0$.

2.2 TRAINING, DECODING AND EVALUATION

Training. To simplify the framework, the slot count K is fixed throughout training. A group holds K *independent* training sequences. In the middle layers, their states are combined before attention to reduce computation, while each sequence retains its own MLP and next-token target.

The objective is unchanged from ordinary language modelling. The same LM head is applied to the final hidden states of every sequence to produce their next-token distributions. The loss is the mean next-token cross-entropy over all GKT positions, so every slot contributes equally:

$$\mathcal{L}_{\mathrm{LM}} = \frac{1}{GKT} \sum_{g=1}^{G} \sum_{s=1}^{K} \sum_{t=1}^{T} -\log p_{g,s,t}(y_{g,s,t}). \tag{12}$$

Optimizer, learning rates and schedule are inherited from the baseline recipe (Section 3), with one exception: the multiplexer and demultiplexer parameters use a separate AdamW group (Loshchilov & Hutter, 2019) with their own base learning rate. This group follows the same warmup and warmdown schedule as the rest of the model.

Decoding. One forward pass advances all K sequences in a group and produces a separate next-token distribution for each. The ordinary layers and MLP computation remain separate for every sequence, while each middle layer computes attention once for the group. Thus each decoding step emits K raw tokens.

The key/value cache follows the two attention paths. The ordinary layers attend over each sequence's own raw history, so they cache K rows per group. The middle layers attend over the shared history, so they cache one row per group rather than K.

The group is the unit of execution, so K sequences advance in lockstep. The multiplexer forms the group's attention input from their states, so arbitrary filler in an idle slot can perturb the other sequences. We distinguish two cases. If a group has fewer than K sequences, available sequences are reused in the remaining slots, and outputs from those copies are ignored. If a sequence ends before its group-mates, its unscored tail is filled by cycling through real tokens from that same sequence (the `cycle` tail policy). Both choices keep the group populated without introducing unrelated examples.

A more systematic alternative would expose the model during training to variable numbers of active slots and the same tail patterns used at inference. We do not use this extra curriculum in the main runs for simplicity.

Evaluation. We evaluate pretraining quality with validation bits per byte (BPB), computed as the summed next-token negative log-likelihood divided by the number of UTF-8 bytes in the target tokens and by $\ln 2$. For downstream behavior, we use nanochat's implementation of the DCLM CORE suite (Karpathy, 2025; Li et al., 2024) and report the CORE score. Unless stated otherwise, each batch-multiplexed group contains K distinct evaluation sequences, and each sequence is scored only on its own target tokens. We refer to this as packed evaluation, in contrast to replicating the same sequence across all K slots. The unscored tail of a shorter sequence is filled using the `cycle` policy described above.

3 SCALING EXPERIMENTS

Because language-model quality varies systematically with model scale and token budget (Kaplan et al., 2020; Hoffmann et al., 2022), we compare the two approaches as model families across depth and training horizon rather than relying on a single pair of checkpoints. We first restate nanochat's model scaling recipe, which serves as our baseline, and then define the corresponding scaling recipe of the batch-multiplexed model family.

3.1 SCALING OF BASELINE MODELS

For the baseline, we use nanochat's model family and its empirically fitted compute-optimal training recipe (Karpathy, 2025). The main scaling runs use ClimbMix for pretraining (Diao et al., 2025). Depth is the only dial; everything else is derived:

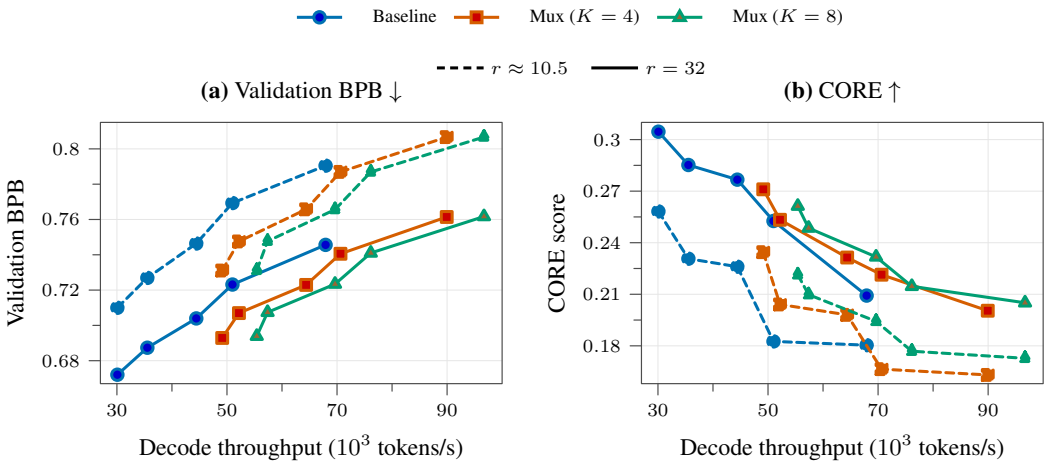


Figure 2: Validation quality against measured decode throughput at two training horizons. Panel (a) reports validation BPB and panel (b) reports CORE. Colors and markers identify baseline, $K = 4$, and $K = 8$ models; dashed curves use $r \approx 10.5$ and solid curves use $r = 32$. Each curve contains depths $D \in \{16, 18, 20, 22, 24\}$; depth decreases from left to right. All endpoints use the matched training and validation protocol. Lower BPB and higher CORE are better.

Shape. Width follows $d = 128\lceil D \cdot 64/128 \rceil$, an aspect ratio of 64 rounded to the head dimension.

Token budget. The horizon is $D_{\text{tok}} = r\,N$, with N counting projection matrices and the LM head and excluding all embeddings. This follows nanochat's parameter-count convention. Nanochat's empirical scaling fit gives $r \approx 10.5$, which defines one comparison horizon. We also use $r = 32$, roughly $3\times$ as many training tokens at fixed model size, to test the same quality–throughput trade-off in the over-trained regime.

Batch size and weight decay. Batch follows $B = 2^{19}(D_{\text{tok}}/D_{\text{ref}})^{0.383}$ clamped to a power of two, and weight decay follows $\lambda = \lambda_0\sqrt{B/B_{\text{ref}}}\,(D_{\text{ref}}/D_{\text{tok}})$, where D_{ref} is a calibration horizon fixed by convention and independent of the run's own length. Optimizer, learning rates, warmup, warmdown fraction and numerical precision are untouched.

3.2 Scaling of batch-multiplexed models

We fix the first and last four blocks as ordinary nanochat blocks. At total depth D, the remaining $D - 8$ middle blocks use the batch-multiplexed attention described in Section 2. The backbone shape matches the nanochat baseline at the same depth. For each fixed K, depth is the only quantity varied along the scaling curve.

At each depth and training horizon, a batch-multiplexed model uses the raw-token batch size, optimizer step count, and weight decay prescribed for the corresponding nanochat baseline. We do not extend the training horizon for batch-multiplexed models for the extra multiplexer and demultiplexer matrices. Each matched pair therefore sees the same number of raw tokens per step and in total, but the effective batch seen by middle-block attention in a batch-multiplexed model is smaller by a factor of K. Thus the batch-multiplexed models use less training compute than their partners. We also do not convert this compute difference into extra training steps. Because the recipe was fitted for the nanochat baseline, we do not claim that it is compute-optimal for batch-multiplexed models.

3.3 Pareto frontier

We compare quality against empirically measured decode throughput. Every model is trained under the protocol described above and timed under the same decoding protocol. Decode throughput, in

tokens per second, is the horizontal axis. Validation BPB and CORE are separate vertical axes: lower BPB and higher CORE indicate better quality. The curves ask whether batch-multiplexing improves quality at a given decode throughput, or delivers more throughput at a given quality. We report both the $r \approx 10.5$ and $r = 32$ sweeps as main results.

Decode throughput is measured at each final checkpoint on one B200, using the same original batch of 256 sequences and context length 2048. All throughput measurements use BF16, FlashAttention-4 (Zadouri et al., 2026), and CUDA Graph. We report original tokens per second for model decoding, including the LM head but excluding tokenization and sampling.

Across both training horizons, batch multiplexing improves the empirical quality–throughput frontier (Figure 2). Over the quality range shared with the nanochat family, the $K = 4$ models provide an intermediate trade-off and the $K = 8$ models provide the largest throughput gain. Over the overlapping BPB range, linear interpolation of the scaling curves gives 9–31% higher throughput for $K = 4$ and 17–48% for $K = 8$ at $r \approx 10.5$; the corresponding gains at $r = 32$ are 10–28% and 18–42%. CORE is less stable at these small model scales.

4 Further experiments

4.1 Slot count

Larger K allows more independent sequences to share one attention computation, but makes it harder to combine their states into a shared representation and return a distinct attention update back. We test how this choice affects quality and decode throughput. The study includes a baseline model and batch-multiplexed models of the same depth, with $K \in \{1, 2, 4, 8\}$. Note that the $K = 1$ model still retains the multiplexing mechanism.

The D24 point from Section 3 is the fixed model reference. We spell out its configuration here as a supplement. Total depth is $D = 24$ and model width is $d = 1536$. Attention has 12 query heads and 12 key/value heads, each of width 128; the MLP hidden width is 6144. The vocabulary has 32,768 entries and the sequence length is 2048. The first and last four blocks are ordinary nanochat blocks. In the 16 middle blocks, attention is computed once per group while the residual stream and MLP remain separate for every sequence. The whole model follows nanochat's SSSL attention-window pattern (Karpathy, 2025): three 512-token windows followed by one 2048-token window, repeated through depth. All batch-multiplexed models retain the same multiplexer and demultiplexer design: each middle block partitions its d channels into $J = 32$ ranges of 48 channels, as in Equation 8, and the demultiplexer uses a four-dimensional query and key for each range ($d_q = 4$).

All models follow the matched training and validation protocol from Section 3 at $r = 32$. For D24, this corresponds to 22,272 optimizer steps.

As shown in Figure 3, the $K = 1$ control closely matches the performance of the baseline model but provides no throughput gain. For $K > 1$, larger groups increase decode throughput, while BPB changes little from $K = 2$ to $K = 8$. CORE is less monotonic, with $K = 4$ giving the strongest result among the settings that share attention. We therefore select $K = 4, 8$ as two sweet spots.

4.2 Learned write weights

The multiplexer in Equation 8 assigns a separate write weight to every slot and channel range. We first examine whether the learned weights remain close to uniform mixing. For the D24 model with $K = 8$, we inspect both the individual range-level weights $\omega_{s,\ell,j}$ and their token-level mean across the $J = 32$ ranges,

$$\bar{\omega}_{s,\ell} = \frac{1}{J} \sum_{j=1}^{J} \omega_{s,\ell,j}, \tag{13}$$

where uniform mixing would assign every slot weight $1/K = 0.125$.

The range-level weights depart substantially from uniform mixing (Figure 4): 22.4% are below 0.05, while 10.3% are at least 0.25, twice the uniform value. Thus, for a given channel range, the multiplexer suppresses many slot contributions while assigning substantially more weight to others. This selectivity is especially coherent in layers 7–8: even after averaging over all 32 channel ranges,

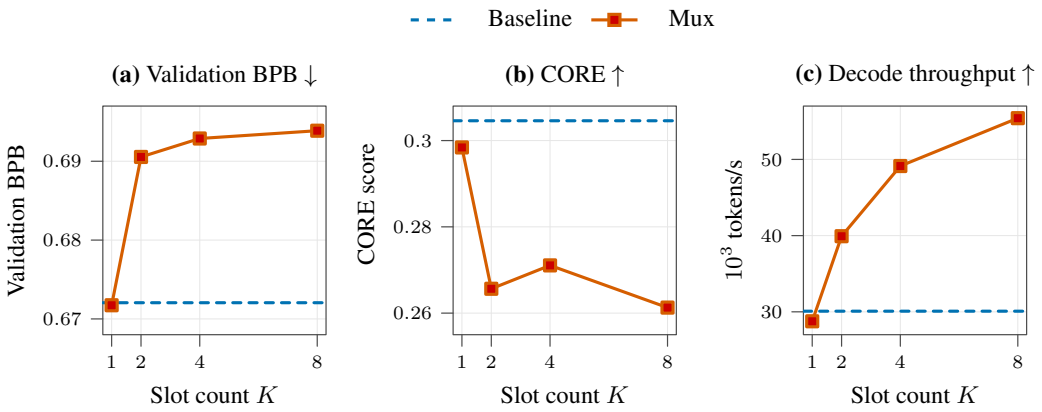


Figure 3: Effect of slot count at fixed D24 architecture under the matched training and validation protocol. The solid curve contains $K \in \{1, 2, 4, 8\}$; dashed horizontal lines mark the D24 reference baseline. Lower BPB and higher CORE and throughput are better.

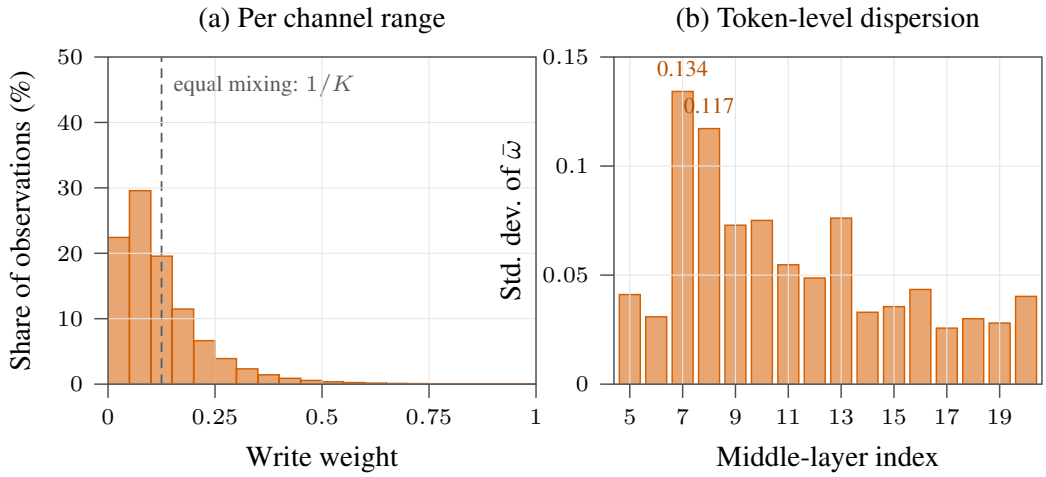


Figure 4: Learned write-weight distributions for the D24 batch-multiplexed model with $K = 8$. Panel (a) pools channel-range weights across the 16 middle layers and eight slots; the dashed line marks uniform mixing, $1/K$. Panel (b) reports the standard deviation of token-level mean weights by middle layer after averaging over 32 channel ranges.

the standard deviation across slots is 0.134 in layer 7 and 0.117 in layer 8, compared with 0.026–0.030 in layers 17–19. This contrast suggests that some layers implement a pronounced sequence-level selection mechanism, whereas at other depths range-specific preferences largely cancel when aggregated.

We next ask whether these learned preferences identify which tokens are difficult to predict. Within each group, position, and middle layer, we rank the eight slots by $\bar{\omega}_{s,\ell}$, then pool the ranks across all 16 middle layers and measure the next-token loss of the corresponding slot. As a control, we evaluate the same tokens with the matched D24 nanochat model.

As shown in Figure 5, next-token loss increases monotonically with write-weight rank. From rank 1 to rank 8, loss rises from 1.784 to 2.630, while the reference loss of the baseline model rises from 1.726 to 2.552. The curves are nearly parallel. The score therefore appears to track intrinsic prediction difficulty and give harder tokens greater influence on the shared attention input, rather than mixing the slots uniformly. Together with Figure 4, this shows that the weights are structured: they specialize across channel ranges and consistently give harder tokens greater influence.

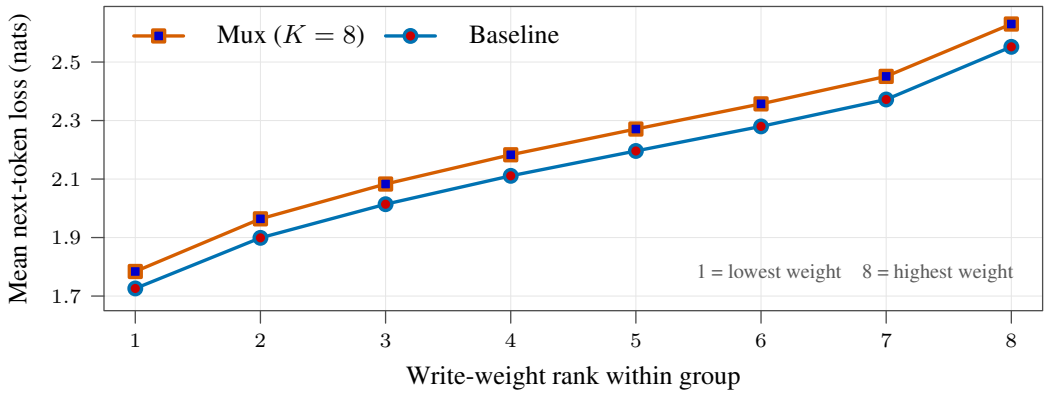


Figure 5: Next-token loss versus within-group write-weight rank for the D24 batch-multiplexed model with $K = 8$. Ranks are computed separately within each group, position, and middle layer after averaging over 32 channel ranges. The nanochat reference evaluates the same tokens with the matched D24 checkpoint. Each point averages 7,739,712 cases.

Further experiments are provided in the appendix.

5 Related work

Efficient attention and KV caches. Attention architectures have progressively reduced KV redundancy across different dimensions. Standard multi-head attention gives each query head its own key and value projections (Vaswani et al., 2017). Multi-query attention shares one set of key-value heads across all query heads (Shazeer, 2019), while grouped-query attention shares within groups of query heads (Ainslie et al., 2023). Multi-head latent attention instead compresses the key-value representation into a low-dimensional latent cache (DeepSeek-AI, 2024). These designs reduce KV-cache redundancy within a sequence, but each request still maintains a separate history.

Long-context methods act along the sequence dimension by reducing the history entries read by each query. Sparse Transformer uses structured attention patterns (Child et al., 2019), while Routing Transformer selects tokens through content-dependent clustering (Roy et al., 2021). DeepSeek Sparse Attention uses a learned indexer to retrieve a small set of preceding KV entries (DeepSeek-AI, 2025). DeepSeek-V4's Compressed Sparse Attention first combines local groups of KV entries and then performs sparse selection, whereas Heavily Compressed Attention combines larger groups and attends densely to the shortened history (DeepSeek-AI, 2026). Batch multiplexing acts along a different axis: it combines same-position states from independent sequences into one attention input and then separates the resulting update.

Sharing across autoregressive sequences. Some serving methods accelerate exact inference when multiple generations share cached prefix states, without changing the model. Bifurcated Attention separates incremental attention over a common prefill cache from attention over each sequence's private decoding history, reducing redundant KV-cache reads (Athiwaratkun et al., 2024). Hydragen combines this decomposition with inter-sequence batching and extends it to hierarchical prefix-sharing patterns (Juravsky et al., 2024). These methods require actual overlap between requests' KV states. Batch multiplexing instead learns to share selected attention computations among sequences whose histories may be unrelated.

Superposed Decoding generates multiple drafts from a common prompt with an unmodified model by feeding a weighted combination of their latest token embeddings through one forward pass at each decoding step (Shen et al., 2024). It uses draft-specific n-gram scores to select continuations rather than maintaining a separate language-model state for every draft. Batch multiplexing instead preserves a separate residual state and next-token distribution for every sequence, including when the grouped requests are unrelated.

Data multiplexing and computation in superposition. DataMUX combines several inputs using fixed linear transformations, processes the mixed representation with one network, and demultiplexes its output (Murahari et al., 2022). MUX-PLMs extend this approach to pretrained encoder models, using multiplexed pretraining and downstream fine-tuning (Murahari et al., 2023). RevMUX instead trains reversible multiplexing adapters for classification while holding the backbone fixed during adapter training (Xu et al., 2024). These methods obtain predictions for each multiplexed input group in one model pass; they do not address the need to maintain multiple evolving histories across repeated autoregressive decoding steps.

MIMOFormer uses fixed binding keys to compute Performer-style linear attention in superposition and includes a variant that separates the channels again before the per-input MLP (Menet et al., 2023). Its experiments focus on non-autoregressive Long Range Arena tasks. Batch multiplexing instead uses learned multiplexing and demultiplexing around ordinary causal attention in selected blocks of a decoder-only language model.

6 Limitations and conclusion

Limitations. The present study should be read as a first scaling test of batch multiplexing, rather than an exhaustive search over the design space. The training recipe was inherited from the nanochat baseline and may not be optimal for batch-multiplexed models. Several design choices, including the placement of shared-attention blocks and the multiplexer and demultiplexer architectures, remain heuristic and have not been systematically validated, leaving room for further optimization. Finally, batch multiplexing operates along a different design axis from other decoding-level optimizations and is therefore compatible with them in principle; whether their benefits compose in practice still requires experimental validation.

Conclusion. Batch multiplexing shares attention across independent autoregressive sequences along the batch dimension while preserving a separate residual stream, MLP, and next-token distribution for each sequence. Under matched training and evaluation protocols, the $K = 4$ and $K = 8$ model families improve the empirical quality–throughput frontier over nanochat at both $r \approx 10.5$ and $r = 32$. The slot-count and write-weight analyses further indicate that shared attention is not obtained by simply averaging sequences: the multiplexer learns structured, depth-dependent weights that give intrinsically harder tokens greater influence. Together, these results establish cross-sequence attention sharing as a promising direction for more efficient autoregressive decoding.

Future work. Several directions remain open. At the model and system level, the quality–throughput trade-off should be tested at larger scales and under more demanding real-time serving workloads. Multi-user deployment also requires a direct privacy analysis: when sequences share attention, can information from one request be inferred from another user's outputs? At the training level, it remains to be seen whether an existing pretrained checkpoint can be adapted to batch multiplexing through continued pretraining, instead of training a multiplexed model from scratch. Batch multiplexing may also benefit sample-heavy post-training methods such as GRPO (Shao et al., 2024), where generating independent rollouts is a major cost. Finally, the same principle could be explored in models beyond language modeling.

Reproducibility statement

The model architecture, multiplexing and demultiplexing operations, training objective, decoding policy, and evaluation metrics are specified in Section 2. Section 3 defines the baseline and batch-multiplexed model families, token-budget conventions, optimization settings, and decode-throughput protocol. Section 4 reports the fixed configurations used in the controlled analyses. The appendix documents its additional experiments with the same level of configuration and evaluation detail. The accompanying anonymous code release provides the model and evaluation implementations, exact configuration and launch files for the reported runs, data-preparation instructions, and the scripts and exported measurements used to reproduce the reported tables and figures. All training and evaluation datasets are publicly available and cited in the paper.

AI USE STATEMENT

Generative AI tools were used to assist with implementing and debugging parts of the experimental code, checking experiment configurations, preparing analysis and plotting scripts, and polishing parts of the manuscript. The core research idea, methodological design, experimental objectives, and final evaluation protocols were developed and determined by the authors. The authors reviewed and tested all AI-assisted code, verified the reported numbers against the corresponding training and evaluation records, and reviewed and revised all AI-assisted text and figures. All reported results come from executed experiments, and the authors take full responsibility for the content of this work.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, 2023. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298/.

Ben Athiwaratkun, Sujan Kumar Gonugondla, Sanjay Krishna Gouda, Haifeng Qian, Hantian Ding, Qing Sun, Jun Wang, Jiacheng Guo, Liangfu Chen, Parminder Bhatia, Ramesh Nallapati, Sudipta Sengupta, and Bing Xiang. Bifurcated attention for single-context large-batch sampling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 1971–1991, 2024. URL https://proceedings.mlr.press/v235/athiwaratkun24a.html.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning (ICML)*, 2023.

Cerebras. SlimPajama: A 627b token cleaned and deduplicated version of RedPajama. https://huggingface.co/datasets/cerebras/SlimPajama-627B, 2023.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019. URL https://arxiv.org/abs/1904.10509.

DeepSeek-AI. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026. URL https://arxiv.org/abs/2606.19348.

Shizhe Diao, Yu Yang, Yonggan Fu, Xin Dong, Dan Su, Markus Kliegl, Zijia Chen, Peter Belcak, Yoshi Suhara, Hongxu Yin, Mostofa Patwary, Yingyan Lin, Jan Kautz, and Pavlo Molchanov. CLIMB: Clustering-based iterative data mixture bootstrapping for language model pre-training. *arXiv preprint arXiv:2504.13161*, 2025. URL https://arxiv.org/abs/2504.13161.

Alex Henry, Prudhvi Raj Dachapally, Shubham Shantaram Pawar, and Yuxuan Chen. Query-key normalization for transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 4246–4253, 2020. doi: 10.18653/v1/2020.findings-emnlp.379. URL https://aclanthology.org/2020.findings-emnlp.379/.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2203.15556.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

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

Andrej Karpathy. nanochat: The best ChatGPT that $100 can buy. https://github.com/karpathy/nanochat, 2025.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, et al. DataComp-LM: In search of the next generation of training sets for language models. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 37, 2024. URL https://papers.nips.cc/paper_files/paper/2024/hash/19e4ea30dded58259665db375885e412-Abstract-Datasets_and_Benchmarks_Track.html.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Chris Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Lightning AI. LitGPT. https://github.com/Lightning-AI/litgpt, 2023.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Nicolas Menet, Michael Hersche, Geethan Karunaratne, Luca Benini, Abu Sebastian, and Abbas Rahimi. MIMONets: Multiple-input-multiple-output neural networks exploiting computation in superposition. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023. doi: 10.52202/075280-1718. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/7c7a12559be4501f70d221352514397c-Abstract-Conference.html.

Vishvak Murahari, Carlos E. Jimenez, Runzhe Yang, and Karthik Narasimhan. DataMUX: Data multiplexing for neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022. doi: 10.52202/068431-1273. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/6fc46679a7ba2ec82183cf01b80e5133-Abstract-Conference.html. arXiv:2202.09318.

Vishvak Murahari, Ameet Deshpande, Carlos E. Jimenez, Izhak Shafran, Mingqiu Wang, Yuan Cao, and Karthik Narasimhan. MUX-PLMs: Data multiplexing for high-throughput language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 4540–4554, 2023. doi: 10.18653/v1/2023.findings-emnlp.301. URL https://aclanthology.org/2023.findings-emnlp.301/. arXiv:2302.12441.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021. doi: 10.1162/tacl_a_00353. URL https://aclanthology.org/2021.tacl-1.4/.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019. URL https://arxiv.org/abs/1911.02150.

Ethan Shen, Alan Fan, Sarah M. Pratt, Jae Sung Park, Matthew Wallingford, Sham M. Kakade, Ari Holtzman, Ranjay Krishna, Ali Farhadi, and Aditya Kusupati. Superposed decoding: Multiple generations from a single autoregressive inference pass. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024. doi: 10.52202/079017-3774. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/d74f9efa1d8ca30b31d65cef8de7c2bf-Abstract-Conference.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Yige Xu, Xu Guo, Zhiwei Zeng, and Chunyan Miao. RevMUX: Data multiplexing with reversible adapters for efficient LLM batch inference. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 22072–22087, 2024. doi: 10.18653/v1/2024.emnlp-main.1232. URL https://aclanthology.org/2024.emnlp-main.1232/. arXiv:2410.04519.

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. *arXiv preprint arXiv:2603.05451*, 2026. URL https://arxiv.org/abs/2603.05451.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/1e8a19426224ca89e83cef47f1e7f53b-Abstract.html.

Zhanchao Zhou, Tianyi Wu, Zhiyun Jiang, Fares Obeid, and Zhenzhong Lan. Value residual learning for alleviating attention concentration in transformers. *arXiv preprint arXiv:2410.17897*, 2024. URL https://arxiv.org/abs/2410.17897.

A SLIDING-WINDOW ATTENTION

The main experiments already use nanochat's SSSL window pattern (Section 4.1), showing that batch multiplexing can operate with sliding-window attention. Here we directly test whether replacing full attention with SSSL incurs a larger BPB penalty for a multiplexed model than for its nanochat baseline. We compare full causal attention with SSSL at fixed depth 16. The full-attention models can attend over the entire 2048-token context in every layer. SSSL instead uses three 512-token windows followed by one 2048-token window, repeated across the whole model. Within each pair, the attention window is the only changed setting; the data, training protocol, and validation protocol are unchanged.

Table 1: Validation BPB for matched full-attention and SSSL window runs under the same protocol.

Model	Full attention BPB	SSSL BPB	SSSL − Full
Nanochat D16	0.745128	0.745615	+0.000487
Mux D16	0.781167	0.782019	+0.000852

Both penalties are below 0.001 BPB: 0.000487 for nanochat and 0.000852 for the multiplexed model. In this fixed-depth control, we therefore do not observe a material amplification of the quality cost of sliding-window attention.

B TRANSFER EXPERIMENTS

B.1 DATASET TRANSFER

To test whether our results depend on the training corpus, we repeat the D24 slot-count study from Section 4.1 using the LitGPT TinyLlama data stack (Lightning AI, 2023). This stack combines SlimPajama with its GitHub subset removed and StarCoderData at weights 0.693584 and 0.306416 (Cerebras, 2023; Li et al., 2023); validation uses the filtered SlimPajama validation set. The baseline model and batch-multiplexed models with $K \in \{1, 2, 4, 8\}$ retain the architecture, nanochat tokenizer, and training and evaluation protocols from Section 4.1.

Table 2: Dataset-transfer results at the final 22,272-step checkpoint. All variants use the same $r = 32$ raw-token budget.

Model	Validation BPB ↓	CORE ↑
Baseline D24	0.824896	0.2394
Mux D24, $K = 1$	0.825564	0.2433
Mux D24, $K = 2$	0.851234	0.2078
Mux D24, $K = 4$	0.850014	0.2106
Mux D24, $K = 8$	0.851036	0.2053

The new corpus reproduces the pattern of the previous D24 study. The $K = 1$ control remains within 0.001 BPB of the baseline and slightly improves CORE. For $K \in \{2, 4, 8\}$, the BPB gap is 0.025–0.026 and the CORE gap is 0.029–0.034. These differences are the same order as in the original D24 comparison, indicating that the effect of sharing attention is not specific to the ClimbMix training distribution.

B.2 TRANSFORMER BACKBONE TRANSFER

To test whether the method depends on nanochat's ResFormer-style block (Zhou et al., 2024), we implement it in a native Pythia-1.4B backbone (Biderman et al., 2023). The model design follows Pythia's native D24/H2048 configuration. The baseline model has 24 ordinary Pythia blocks. The batch-multiplexed models place 16 shared-attention blocks between four ordinary blocks at each end, with $K \in \{1, 2, 4, 8\}$. In a shared block, only attention is computed once per group; the native Pythia MLP and parallel residual remain separate for every sequence. The multiplexer and

demultiplexer use the same $J = 32$ channel ranges and $d_q = 4$ read queries and keys as the nanochat experiments.

All variants use the corpus from Section B.1. The Pythia backbone uses the LitGPT (Lightning AI, 2023) AdamW configuration with learning rate 3×10^{-4}, weight decay 0.1, 40 warmup steps, and cosine decay; multiplexer and demultiplexer parameters use a separate AdamW parameter group with base learning rate 3×10^{-5}. All variants match the raw-token budget of the $r = 32$ D24 studies and are evaluated after 22,272 steps.

Table 3: Pythia-1.4B backbone-transfer results at the final 22,272-step checkpoint under the matched raw-token budget.

Model	Validation BPB ↓	CORE ↑
Baseline Pythia	0.902355	0.1972
Mux Pythia, $K = 1$	0.903070	0.1745
Mux Pythia, $K = 2$	0.920685	0.1729
Mux Pythia, $K = 4$	0.926868	0.1691
Mux Pythia, $K = 8$	0.933020	0.1537

The Pythia transfer shows the same qualitative progression. The $K = 1$ model matches the baseline in BPB to within 0.001. For $K > 1$, the BPB and CORE gaps relative to the baseline are 0.018–0.031 and 0.024–0.044, respectively, comparable in magnitude to those observed in the nanochat study. The multiplexing mechanism therefore transfers to a different transformer block without a qualitatively new failure mode.

C Extended training dynamics

The main scaling experiments stop at $r = 32$. To examine whether attention sharing remains trainable under substantially longer optimization, we extend the matched D24 baseline and $K = 8$ runs to $r = 128$. We evaluate validation BPB every 500 steps and once more at the final endpoint.

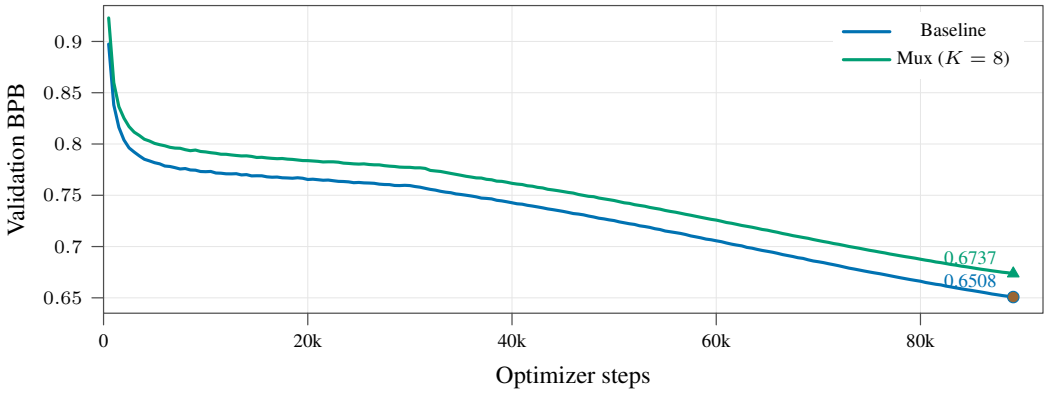


Figure 6: Validation BPB during extended D24 training. The baseline and $K = 8$ model are evaluated every 500 optimizer steps and at the final 89,088-step endpoint.

The two models exhibit similar training dynamics (Figure 6). BPB falls rapidly during the first few thousand steps, improves more slowly over the next stage, and continues to decrease through the end of the extended schedule. The two training curves are nearly parallel, suggesting that most of the gap emerges during the early stage of optimization. The $K = 8$ curve remains above the baseline, but the quality gap does not widen with longer training. The final BPB gap is 0.0218 at $r = 32$ (0.6939 versus 0.6721) and 0.0229 at $r = 128$ (0.6737 versus 0.6508). Thus, additional training improves both models at similar rates rather than causing the batch-multiplexed model to fall progressively further behind.