KERNEL-LEVEL CONTEXT PARALLELISM: BRIDGING INCOMPATIBLE SHARD LAYOUTS IN HYBRID LINEAR-ATTENTION TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Hybrid transformers that interleave softmax attention with linear attention now reach production scale, but under context parallelism (CP) their two layer families require incompatible sequence layouts. Fused softmax-attention CP kernels shard each sequence in a *zigzag* order to balance causal work, whereas state-passing linear-attention kernels require each rank to own a *contiguous* span. A common workaround all-gathers the full hidden state at every linear-attention layer and recomputes the layer redundantly on all P ranks, so its activations and linear-attention compute stay proportional to the full sequence however large P grows. At 256K tokens with P=16, this baseline exhausts an 80 GB H100 before completing a single step. We present KCP, which instead converts between the two layouts with an exactly invertible token permutation. KCP keeps zigzag as the global layout, converts to contiguous order at the entry of each linear-attention layer with a single `all_to_all` of T/P tokens per rank, runs the unmodified native CP kernel, and converts back on exit. The payload of each collective drops by $P\times$; more importantly, per-rank linear-attention compute and activation memory also drop by $P\times$, because no rank recomputes the full sequence. Because the conversion is a pure permutation, its backward pass is the reverse permutation. On a 35B-parameter MoE hybrid across 32 H100 GPUs, KCP achieves 2.5–4.7$\times$ the training throughput of the all-gather baseline at every configuration both can run, and it trains the 256K-token configuration at 53.3 GB peak allocated memory. Training losses agree with the baseline up to floating-point nondeterminism, and the layout conversion accounts for 1.8–2.7% of GPU-busy time across profiled ranks. On a dense hybrid with small bins, where the gather is not the bottleneck, the two methods are within $\pm$7% at three of four points, consistent with gains that come from eliminating full-sequence materialization rather than from a faster kernel.

1 INTRODUCTION

Large language model (LLM) workloads increasingly involve long documents (Bai et al., 2024) and multi-turn agent trajectories that accumulate observations, tool outputs, and actions over tens to hundreds of thousands of tokens (Du et al., 2026). Under causal masking, softmax attention (Vaswani et al., 2017) lets every token attend to its entire prefix, so its cost grows as $O(n^2)$ in the sequence length n. Hybrid linear-attention transformers therefore keep full softmax attention in only a minority of layers and use linear attention, whose cost grows linearly in n, everywhere else (Katharopoulos et al., 2020; Gu & Dao, 2024; Yang et al., 2025). Recent open hybrids place one full-attention layer in every eight (Lieber et al., 2024; MiniMax, 2025) or every four (Kimi Team et al., 2025; Qwen Team, 2026a) layers; the dense Qwen3.5 and Mixture-of-Experts (MoE) Qwen3.6 models we study (Qwen Team, 2026a;b) use the latter ratio.

Hybrid layers reduce the cost of long-context computation, but training on long sequences still requires context parallelism (CP), which shards the token axis across P ranks (Liu et al., 2024; Jacobs et al., 2023). Under CP, the two layer families of a hybrid require incompatible sequence layouts (Figure 1a). Fused softmax-attention CP kernels use a *zigzag* order, in which rank r holds chunks r and $2P-1-r$ of every sequence, to balance causal attention work (Grattafiori et al., 2024; Bu et al.,

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.

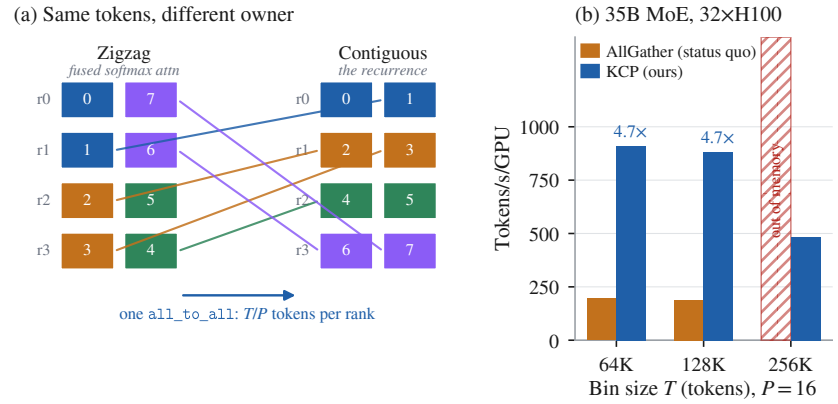

Figure 1: **The layout conflict and what it costs.** *(a)* At $P{=}4$, the same eight chunks of one sequence under the two layouts, one row per rank; color marks each chunk's owner under the contiguous layout, and the lines trace the six chunks that change hands. Fused softmax-attention kernels need zigzag (rank r owns $\{r, 2P{-}1{-}r\}$) for causal load balance; the linear-attention recurrence needs a contiguous span. Neither layout holds more information than the other — only the *owner* of each chunk differs — so the conversion is a permutation, realizable by one `all_to_all` of T/P tokens per rank, where the status quo all-gathers all T tokens to every rank. *(b)* What that costs on a 35B MoE hybrid at $P{=}16$: KCP reaches $4.7\times$ the baseline throughput at both bins the baseline can run, and at 256K the baseline exhausts 80 GB of HBM before finishing one step.

2025). Native state-passing linear-attention kernels instead require each rank to own a *contiguous* span, so that the recurrent state can flow from rank to rank. No single static layout satisfies both.

Because the fused attention kernels fix the global layout to zigzag, the mismatch must be absorbed inside each linear-attention layer. A common workaround, used in the Megatron-based training stack we build on (Shoeybi et al., 2019), all-gathers the full T-token hidden state on every rank, evaluates the linear-attention layer redundantly over the whole packed bin, and keeps only the outputs of the local shard. This defeats the purpose of CP for these layers: their per-rank compute and activation memory scale with the full bin T rather than with the local shard T/P, so increasing P brings no relief. At 256K tokens with $P{=}16$ on 32 H100 GPUs, the baseline exhausts 80 GB of HBM during the first backward pass and cannot complete a single training step (Figure 1b).

We propose KCP (Kernel-level Context Parallelism), which replaces the full-sequence gather with an exactly invertible token permutation. Zigzag and contiguous sharding hold the same tokens and differ only in which rank owns each chunk. KCP therefore builds a bijective redistribution plan from the sequence boundaries of the packed batch and realizes each direction with a single `all_to_all` that moves T/P tokens per rank. At each linear-attention layer, KCP converts the input from zigzag to contiguous order, runs the unmodified native CP kernel on the local shard, and converts the output back. Full-attention layers keep their layout and fused kernels, and the rest of the training loop is unchanged. Because each rank now evaluates only its own shard, per-rank linear-attention compute and activation memory fall by a factor of P.

On a 35B-parameter MoE hybrid trained on 32 H100 GPUs, KCP achieves 2.5–$4.7\times$ the throughput of the all-gather baseline at every configuration both methods complete, and it trains the 256K-token configuration at which the baseline runs out of memory. On a dense 9B hybrid with small 4K–16K bins, where full-sequence materialization is cheap, the two methods are within $\pm 7\%$ at three of four configurations, consistent with gains that come from avoiding materialization rather than from a faster linear-attention kernel. Our contributions are as follows:

- **An invertible layout bridge for hybrid context parallelism.** KCP converts between zigzag and contiguous sharding for variable-length packed batches with one `all_to_all` of T/P tokens per rank in each direction, leaving both the softmax- and linear-attention CP kernels unmodified. Each rank builds its plan on the CPU from `cu_seqlens` without exchanging metadata, and the backward pass is the inverse permutation.

- **A cost analysis that separates three effects.** Relative to the all-gather baseline, KCP reduces the payload of each collective by $P\times$, total communication per layer by $P/2$, and per-rank linear-attention compute and activation memory by $P\times$. The analysis also predicts when the baseline's full-sequence working set sets the memory peak, and our memory measurements match this prediction.
- **An evaluation at production scale with a mechanism control.** On 32 H100 GPUs, we measure throughput, peak memory, loss agreement, and bridge overhead, and we use a small-bin dense control to test whether the gains come from eliminating full-sequence materialization rather than from a different kernel.

2 Background and Related Work

State-passing linear attention. Linear attention (Katharopoulos et al., 2020) avoids the quadratic pairwise-interaction cost of softmax attention. In a simplified unnormalized form, associativity allows the attention computation to be rearranged as $(QK^\top)V = Q(K^\top V)$, so for fixed feature dimensions the cost grows as $O(T)$. Under a causal mask, the computation becomes a left-to-right state update. Let $S_t \in \mathbb{R}^{d_k \times d_v}$ denote the recurrent state after position t; its size depends on the feature dimensions but not on the sequence length. In its simplest gated form (Sun et al., 2023; Dao & Gu, 2024),

$$S_t = \alpha_t S_{t-1} + k_t v_t^\top, \qquad o_t = q_t^\top S_t, \tag{1}$$

where q_t, k_t, and v_t are the query, key, and value at position t, o_t is the output, and the gate α_t controls how much of the previous state is retained. The Gated DeltaNet layers of our target models (Yang et al., 2025) add a delta-rule correction, $S_t = \alpha_t(I - \beta_t k_t k_t^\top)S_{t-1} + \beta_t k_t v_t^\top$. Only one property of these updates matters for parallelization: later positions depend on state propagated from earlier positions. Chunked-parallel kernels process the tokens within a chunk in parallel but still pass the state between chunks, so under their CP interface a rank assigned positions $[a, b)$ must own that interval contiguously and receive the incoming state S_{a-1}. Because a fixed-size state is a lossy summary of the prefix, hybrid models retain some attention layers (Lieber et al., 2024; Ren et al., 2025; Glorioso et al., 2024; De et al., 2024; MiniMax, 2025).

Context parallelism for softmax attention. Fused attention kernels such as FlashAttention avoid materializing the attention matrix, but causal attention still performs quadratic pairwise work (Dao et al., 2022). Context-parallel methods distribute this work by partitioning the token axis. Ring Attention streams key–value blocks around a ring (Liu et al., 2024), DeepSpeed Ulysses uses all-to-all communication to switch between sequence and head sharding (Jacobs et al., 2023), and USP and LoongTrain combine these dimensions for long-context training (Fang & Zhao, 2024; Gu et al., 2024). A contiguous partition is load-imbalanced under a causal mask, because later positions attend to longer prefixes. Striped Attention rebalances the work by permuting tokens across ranks (Brandon et al., 2023), and the zigzag scheme used to train Llama 3 assigns rank r the chunks $\{r, 2P-1-r\}$ of each sequence (Grattafiori et al., 2024); common ring-attention CP implementations use the same zigzag load balancing (Bu et al., 2025; Ghadia et al., 2026).

Sequence parallelism for linear attention. Sequence-parallel methods for linear attention move recurrent states rather than activations, exploiting the fact that the state size is independent of sequence length. LASP passes the state around a ring (Sun et al., 2025b); LASP-2 replaces the ring with an all-gather of intermediate states (Sun et al., 2025a); and ZeCO introduces an All-Scan collective that delivers to each rank exactly the initial state it needs (Chou et al., 2025). The native CP kernels underlying our implementation use this state-exchange design (Yang & Zhang, 2024), which we take as an existing component rather than a contribution. The hybrid extension of LASP-2 also redesigns the communication of the standard-attention modules, so that both layer families share one scheme; this presumes that the global layout can be chosen jointly for both families (Sun et al., 2025a).

Positioning and alternative designs. Our setting removes that freedom: fused softmax-attention kernels fix the global layout to zigzag, while the native linear-attention kernels require contiguous spans. Softmax-oriented CP methods do not specify how linear-attention layers should operate under this fixed layout, and linear-attention SP methods assume that they may choose it. KCP bridges the two by converting between the layouts as an information-preserving token permutation; it introduces

neither a new attention kernel nor a new state-passing algorithm. Two other designs could also avoid the full-sequence gather. *Head sharding* in the style of Ulysses (Jacobs et al., 2023) would all-to-all the projected queries, keys, values, and gates so that each rank processes the full sequence for a $1/P$ fraction of the heads. Its per-rank compute and activation memory also scale as T/P, but it requires the number of linear-attention heads per tensor-parallel rank to be divisible by P, which caps the CP degree, and it exchanges per-head tensors whose combined width can exceed the hidden size d that KCP exchanges. *Zigzag-aware state passing* would treat each rank's two chunks as two virtual contiguous ranks and exchange only fixed-size states, in the spirit of LASP-2 and ZeCO. It avoids activation traffic entirely, but it requires kernels that accept many disjoint spans per rank (two per sequence in a packed batch) and, for delta-rule layers, that combine accumulated state-transition operators across them. KCP instead treats both kernels as black boxes and changes only the layout at their boundary, so it places no constraint on head counts and requires no kernel changes.

3 METHOD

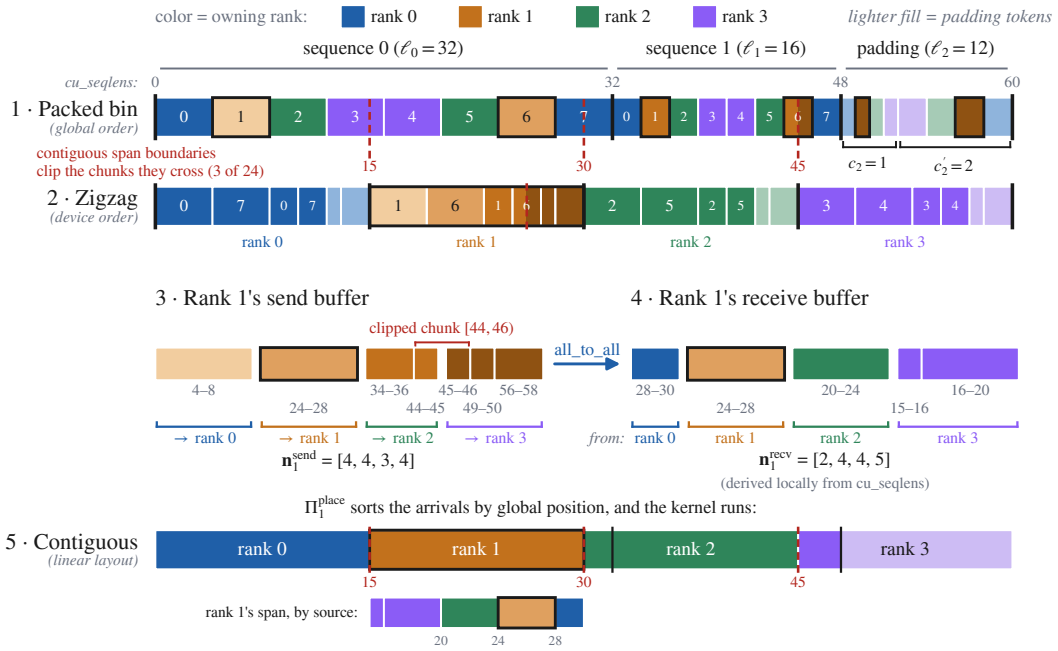

Figure 2: **The KCP redistribution plan.** At P=4, two padded sequences of lengths 32+16 and a 12-token padding tail give T=60 and 15 tokens per rank. The numbered panels trace (1) global packed-bin order, (2) zigzag rank-local shards, (3) packing by destination and global position, (4) all-to-all reception, and (5) placement into contiguous order. The inverse permutation restores zigzag after the linear-attention layer. Appendix A explains the example and padding case in detail.

3.1 PROBLEM SETTING

We consider a packed variable-length batch in which multiple sequences are concatenated along the token axis. We write the resulting hidden states as $x \in \mathbb{R}^{T \times d}$, where T is the total number of tokens in the packed bin and d is the hidden size.

Let m denote the number of packed segments, including an optional non-empty trailing padding pseudo-sequence. For $i \in \{0, \ldots, m-1\}$, let g_i denote the global token offset at which segment i begins. The cumulative-offset vector $(g_0, g_1, \ldots, g_m)$, with $0 = g_0 < g_1 < \cdots < g_m = T$, is stored in `cu_seqlens`. Segment i occupies $[g_i, g_{i+1})$ and has length $\ell_i = g_{i+1} - g_i$.

Let P denote the CP degree and let $r \in \{0, \ldots, P-1\}$ denote a CP rank. The bin length is chosen such that P divides T, so every rank stores exactly T/P token representations. Before packing, each

real sequence is right-padded so that its length is divisible by $2P$. Any remaining capacity at the end of the bin is filled by a loss-masked padding pseudo-sequence. Because T and the padded real sequences are divisible by P, so is the padding tail, though not necessarily by $2P$.

Under the *zigzag* layout, each real sequence i is divided from left to right into $2P$ equal chunks

$$C_{i,0}, C_{i,1}, \ldots, C_{i,2P-1}, \qquad |C_{i,j}| = c_i = \frac{\ell_i}{2P}.$$

Rank r owns chunks $C_{i,r}$ and $C_{i,2P-1-r}$. Pairing an early chunk with a late chunk balances causal softmax-attention work across ranks. The padding pseudo-sequence, whose length may not be divisible by $2P$, uses the generalized split in Appendix A.1.

Under the *contiguous* layout, rank r owns the global packed-bin interval $[rT/P, (r+1)T/P)$. This interval may cross packed-sequence boundaries; the native linear-attention kernels use `cu_seqlens` to reset the recurrent state at those boundaries. Within each sequence, the contiguous layout allows states to be passed from earlier to later positions.

Both layouts assign exactly T/P tokens to every rank and preserve the same token values, logical positions, and sequence boundaries. They differ only in rank ownership and local storage order. The goal of KCP is to redistribute tokens between these layouts without replicating the full sequence.

3.2 The redistribution plan

Let $p = T/P$ denote the number of tokens stored by each rank. Because the zigzag and contiguous layouts contain the same tokens and differ only in rank ownership and local order, converting between them requires only a distributed permutation. KCP constructs this permutation once per packed batch from `cu_seqlens` and executes it with a single `all_to_all_single` in each direction.

Plan construction. Let $\mathcal{Z}_r$ denote the set of global token positions owned by rank r under zigzag, and let $I_q = [qp, (q+1)p)$ be the contiguous span of rank q. The tokens sent from source rank r to destination rank q are precisely the non-empty interval pieces in $\mathcal{Z}_r \cap I_q$. Let $n_{r\to q}$ be their total number of tokens, and define

$$\mathbf{n}_r^{\text{send}} = (n_{r\to 0}, \ldots, n_{r\to P-1}), \qquad \mathbf{n}_r^{\text{recv}} = (n_{0\to r}, \ldots, n_{P-1\to r}).$$

On the send side, rank r orders its pieces by destination rank and then by global position. This order is represented by a local packing permutation matrix Π_r^{pack}. On the receive side, rank r considers source ranks in order and reconstructs the same interval pieces from the shared `cu_seqlens`. Their global positions define a placement permutation matrix Π_r^{place} that restores contiguous order. Because both sides construct the same intersections from the same sequence boundaries, the split sizes and piece orders agree without exchanging additional metadata (Figure 2).

Applying the plan. We denote the packing, all-to-all, and placement composition by $\mathcal{P}_{\text{zz}\to\text{ctg}}$ and its inverse by $\mathcal{P}_{\text{ctg}\to\text{zz}}$. Each rank constructs and caches its local index and split-size vectors once per batch on the CPU, reusing them across linear-attention layers. Appendix A gives the matrix formulation, construction algorithm, and worked example.

3.3 Integration

Let $\mathcal{F}_{\text{LA}}^{\text{ctg}}$ denote a native linear-attention CP layer whose input and output use contiguous sharding. KCP exposes the same layer to the rest of the model as a zigzag-sharded operator by composing it with the two redistributions:

$$\mathcal{F}_{\text{LA}}^{\text{zz}} = \mathcal{P}_{\text{ctg}\to\text{zz}} \circ \mathcal{F}_{\text{LA}}^{\text{ctg}} \circ \mathcal{P}_{\text{zz}\to\text{ctg}}. \tag{2}$$

The first redistribution gives each rank its contiguous shard of $p = T/P$ tokens. The native kernel then evaluates only that shard and passes its fixed-size recurrent states between ranks. The inverse redistribution restores the original zigzag ownership and local order before the next layer.

Equation (2) keeps zigzag as the layout at every model layer boundary. Full softmax-attention layers therefore remain unchanged and retain their original fused kernels and causal load balancing. The

conversion is confined to the entry and exit of each linear-attention layer; token values, logical positions, sequence boundaries, and the surrounding optimizer, checkpointing, pipeline schedule, and loss computation are unchanged.

The target linear-attention block also contains a short causal convolution before its recurrent update. Contiguous sharding requires a small halo exchange at rank boundaries for this convolution, with a volume determined by the fixed kernel width rather than by T. This exchange, recurrent-state passing, and state resets at `cu_seqlens` boundaries are already provided by the native CP implementation (Yang & Zhang, 2024); KCP changes only the layout at its boundary. Section 5.2 verifies the complete composition end to end.

4 Correctness and Cost

We state the two properties that KCP relies on; Appendix B gives proofs and details.

4.1 Correctness

Proposition 1 (Exact redistribution)**.** Under the padding conditions of Section 3.1, $\mathcal{P}_{\mathrm{zz}\to\mathrm{ctg}}$ permutes the T token rows of x, the independently built send and receive plans agree on the size and order of every message, and $\mathcal{P}_{\mathrm{ctg}\to\mathrm{zz}} = \mathcal{P}_{\mathrm{zz}\to\mathrm{ctg}}^{-1}$. Consequently, for any loss $\mathcal{L}$, $\nabla_{x^{\mathrm{zz}}}\mathcal{L} = \mathcal{P}_{\mathrm{ctg}\to\mathrm{zz}}\big(\nabla_{x^{\mathrm{ctg}}}\mathcal{L}\big)$.

Proof sketch. The generalized split of Equation (3) partitions every packed segment, so the zigzag ownership sets partition $[0, T)$, as do the contiguous spans. The sender and the receiver of each message derive the same interval pieces $\mathcal{Z}_r \cap I_q$ from `cu_seqlens` and order them by global start, so split sizes and piece orders agree without exchanging metadata. The redistribution is therefore a composition of three permutations. It moves values without arithmetic, so the round trip is bit-exact, and its Jacobian is the same permutation applied to every feature, whose transpose is its inverse. The backward pass thus needs neither a reduction nor numerical scaling. □

4.2 Cost

Proposition 2 (Cost per linear-attention layer)**.** Consider one linear-attention layer on one rank, with $p = T/P$ local tokens, hidden size d, and b bytes per element. *(i)* One KCP all-to-all moves a logical payload of pdb bytes; one all-gather materializes Tdb, a ratio of P. *(ii)* Over the forward pass, backward pass, and full recomputation, the baseline moves $V_{\mathrm{base}} = 3Tdb$ and KCP moves $V_{\mathrm{KCP}} = 6pdb$, a ratio of $P/2$ (also without recomputation). *(iii)* Its sequence-dependent compute and activation memory fall from $\Theta(T)$ and $\Theta(Td)$ to $\Theta(T/P)$ and $\Theta(Td/P)$.

The ratio in (ii) is $P/2$ rather than P because KCP crosses the layout boundary twice per traversal, whereas the baseline gathers once. These are logical volumes; actual network traffic depends on the collective algorithm and topology. Proposition 2 bounds only the linear-attention terms: full-attention layers, MoE blocks, and the bridge itself remain, so it does not predict a $P\times$ end-to-end speedup.

It does make a testable prediction about memory. Every other sequence-dependent per-rank term shrinks as T/P, whereas the baseline's full-sequence working set (gathered input and recomputed activations) depends on T alone. At small P both methods should therefore reach the same peak, set by the sharded terms they share. Once P is large enough for that term to set the baseline's peak, the baseline should exceed KCP by an amount that grows linearly with T, which Section 5.2 tests.

5 Experiments

We describe the setup, report throughput, memory, and numerical agreement, and then analyze the overhead of the bridge and the regime in which KCP helps.

5.1 Experimental Setup

Models and hardware. We use two open-weight hybrid LLMs with one full-attention layer in every four (Table 4 in Appendix C.1): Qwen3.6-35B-A3B (Qwen Team, 2026b) for the MoE experiments

Table 1: **Main throughput and memory results.** Tokens/s/GPU and peak allocated memory (GB) are reported for the 35B MoE run on 32×H100 and the 9B dense control on 8×H100. The configuration column gives the bin size T and CP degree P; peak-memory entries are ALLGATHER/KCP. Speedup is the ratio of aggregate throughputs; both arms process identical tokens at every step (Section 5.1). Settings marked † have per-GPU shards of 32K tokens or more, twice the largest shard either method trained, and both methods run out of memory.

Model	Config (T/P)	ALLGATHER tok/s/GPU	KCP tok/s/GPU	Speedup	Peak GB ALLGATHER / KCP
Qwen3.6-35B-A3B (MoE)	64K / 4	499	1,245	2.49×	52.7 / 52.7
	64K / 8	273	1,270	4.66×	36.3 / 36.3
	64K / 16	195	909	4.67×	32.1 / 28.1
	128K / 4†	OOM	OOM	—	— / —
	128K / 8	247	636	2.58×	53.0 / 53.0
	128K / 16	189	881	4.66×	44.9 / 36.6
	256K / 4†	OOM	OOM	—	— / —
	256K / 8†	OOM	OOM	—	— / —
	256K / 16	**OOM**	**484**	**KCP only**	**— / 53.3**
Qwen3.5-9B (dense)	4K / 2	736	681	0.93×	54.11 / 54.11
	8K / 2	1,668	1,689	1.01×	54.15 / 54.16
	8K / 4	646	681	1.05×	54.14 / 54.14
	16K / 4	837	1,255	1.50×	54.20 / 54.20

and Qwen3.5-9B (Qwen Team, 2026a) for the dense ones. MoE runs use 4 nodes of 8×H100 80 GB SXM (32 GPUs, NVLink intra-node, InfiniBand across) at TP1·PP2·EP8 with $P \in \{4, 8, 16\}$; dense runs use one such node at TP2·PP1 with $P \in \{2, 4\}$. **Full activation recomputation is on throughout**, matching the accounting of Proposition 2.

Data. We train on multi-turn search-agent trajectories (Du et al., 2026). It contains 11,677 rows totaling approximately 793 million tokens, with a median of 62,973 and a maximum of 368,342 tokens per row. All runs use these real trajectories, packed whole into bins; row lengths span every bin size we evaluate.

Controlling the token budget. We declare the global bin size T and derive the per-GPU budget as T/P. To prevent oversized packed rows from confounding this sweep, configurations are drawn from length buckets whose rows fit the target (Appendix C.2).

Metrics. Each throughput configuration runs for 30 steps. Throughput is the total number of tokens divided by the total training time over the last 25 steps; the first five steps, which include Triton autotuning of the linear-attention kernels, are dropped for both arms. Both arms process identical tokens at every step, so throughput ratios are wall-time ratios. Step times are bimodal on both arms (Appendix C.4). We therefore aggregate over all post-warmup steps, which weights each mode by how often it occurs, rather than taking a median, which would report only one mode; Appendix C.4 also reports median-based ratios. Peak memory is the logged peak of allocated GPU memory.

5.2 MAIN RESULTS

KCP improves throughput and trains a configuration the baseline cannot. Table 1 compares the two methods when training Qwen3.6-35B-A3B on 32 H100 GPUs with 64K, 128K, and 256K bins and $P \in \{4, 8, 16\}$; Figure 6 in Appendix C.3 shows the same results as bars. Across all five configurations that both methods complete, KCP achieves 2.49–4.67× the baseline throughput. At 256K/P=16, the baseline runs out of memory during the first backward pass, whereas KCP trains at 484 tokens/s/GPU with 53.3 GB of peak allocated memory. Two repeats reproduce this outcome: one matches the throughput to within 0.2% (Appendix C.6), and a separate 10-step run peaks at 52.8 GB.

Step times are bimodal on both arms, and the baseline spends many more post-warmup steps in its slow mode. At 128K/P=16, the baseline alternates between about 20 and 98 s/step through the final step and is slow (more than twice its fastest step) in 16 of 25 steps, whereas KCP stays near 10 s/step except for two long steps, which we count (Figure 3). Across the five MoE configurations, the

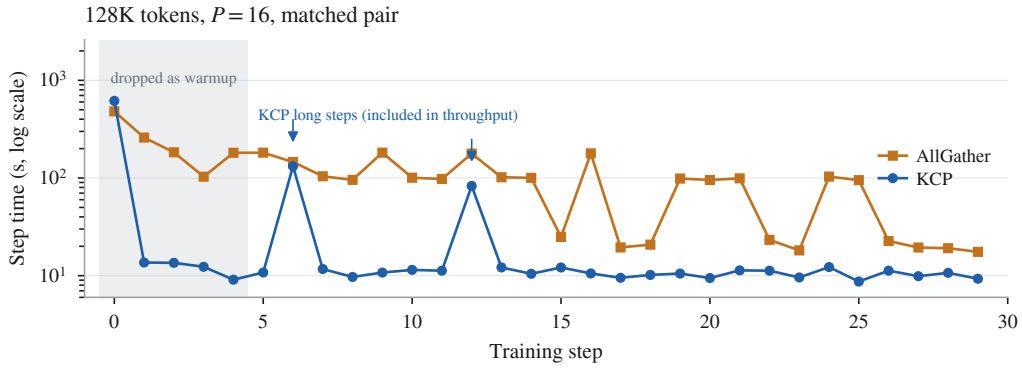

Figure 3: Per-step wall-clock time for the Qwen3.6-35B-A3B runs at 128K tokens and P=16 (log scale). The gray region marks the five warmup steps. After warmup, the baseline alternates between ∼20 and ∼98 s/step through the final step, whereas KCP stays near ∼10 s/step apart from two long steps, which are included in the reported throughput.

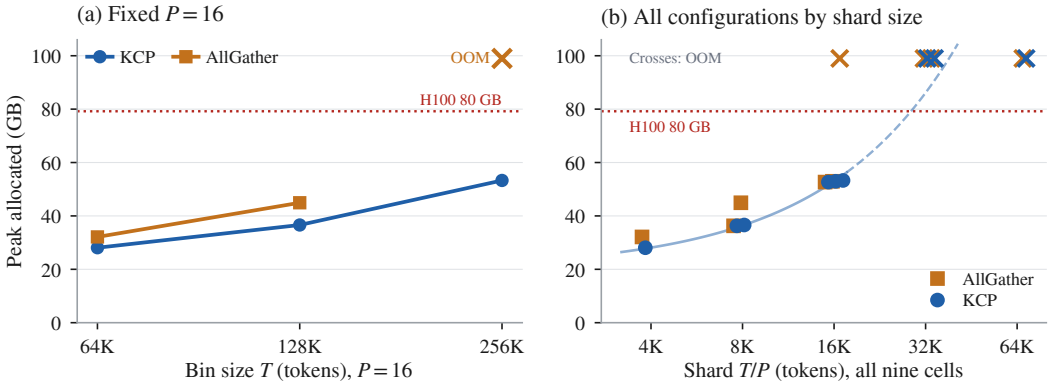

Figure 4: Peak allocated memory of the 35B MoE runs. *(a)* At P=16, the gap between the methods doubles from 64K to 128K, and the baseline runs out of memory at 256K. *(b)* All nine configurations against the shard size T/P: KCP points with equal shards nearly coincide (line: linear fit, dashed where extrapolated), whereas the baseline also depends on T. Crosses mark out-of-memory runs.

baseline is slow in 9–24 of 25 steps and KCP in 1–10 (Table 7). The comparison does not hinge on the choice of statistic: ratios of median step times also favor KCP at every configuration (1.8–9.1×).

Peak memory follows the prediction of Section 4.2. At $P \in \{4, 8\}$, the two methods peak within 0.01 GB of each other, consistent with a peak set by the sharded terms they share. At P=16, KCP is lower by 4.0 GB at 64K and by 8.3 GB at 128K: the gap doubles with T, as expected once the baseline's full-sequence term sets its peak. Figure 4 shows both effects.

KCP preserves the training trajectory within run-to-run variation. We first test the redistribution plan on CPU, using random `cu_seqlens` and every (P, r). We then compare the two training paths end to end with shuffling disabled and identical initialization. Both paths therefore receive bit-identical data, and any subsequent difference can only arise from floating-point reassociation, nondeterministic reduction order, MoE routing, or kernel autotuning. As reported in Table 2, the relative loss difference at step 0 is 1.9×10^{-4} for the dense model and 1.8×10^{-4} for the MoE model. Over all steps, the mean absolute cross-arm difference for the MoE model is 5.1×10^{-4}, below the 8.8×10^{-4} that separates two runs of the same code (Figure 5). The dense result is reported as an absolute difference because its loss approaches zero during the run. Together, these results support agreement up to floating-point reassociation rather than bitwise equality.

Table 2: Loss agreement, ALLGATHER vs. KCP, shuffling disabled, 60 steps. Step 0 is quoted separately because weights are still bit-identical there. The cross-arm columns compare ALLGATHER and KCP; the same-code mean is the run-to-run reference used as the noise floor.

Model	Config	Step-0 rel.	Mean $\lvert\Delta\rvert$	Max $\lvert\Delta\rvert$	Noise floor
Qwen3.5-9B (dense)	4K, P=2	1.9e−4	2.2e−3	1.9e−2	—
Qwen3.6-35B-A3B (MoE)	64K, P=8	1.8e−4	5.1e−4	1.7e−3	8.8e−4

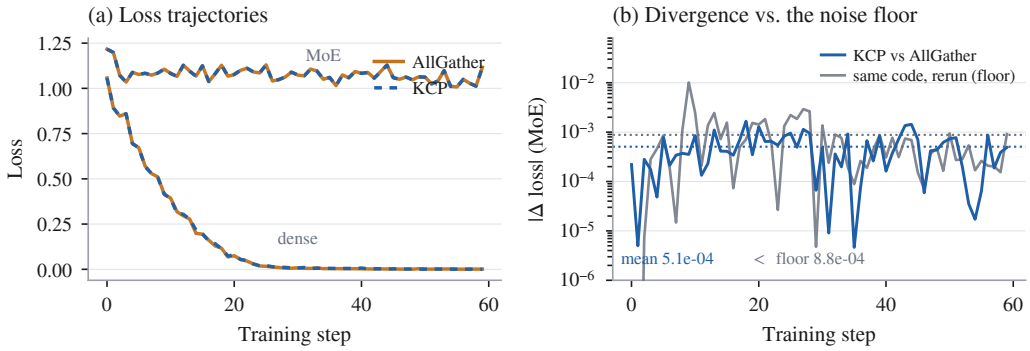

Figure 5: Loss agreement over 60 steps. *(a)* The trajectories are visually indistinguishable for both families. *(b)* Absolute loss difference for the MoE model: on average, the cross-arm difference (blue) is below the difference between two runs of the same code (gray). Dotted lines mark the two means.

5.3 ANALYSIS AND ROBUSTNESS CHECKS

Layout-bridge overhead. We profile the 35B MoE model at 256K/P=16 on 32 H100 GPUs. On rank 0, the trace contains 97.2 s of GPU-busy time over two profiled rollouts, of which the KCP bridge accounts for 1.8%; across six sampled ranks spanning both pipeline stages, the share is 1.8–2.7% (median 2.1%). Most of the GPU-busy time on rank 0 (86%) is NCCL send/receive and all-gather traffic outside the bridge, which includes time spent waiting on peers, so this percentage understates the bridge's cost relative to computation: its 1.7 s compares with 3.8 s in linear-attention kernels and 3.6 s in FlashAttention, and it is smaller than the 2.2 s spent in the MoE dispatch all-to-all (Appendix C.5). The 256K configuration also remains feasible with an optimized MoE dispatcher and when the same 32 GPUs are repartitioned as TP1·PP1·CP32 (Appendix C.6).

KCP provides little benefit when full-sequence materialization is inexpensive. As a control, we train dense Qwen3.5-9B on one 8-GPU H100 node with TP2, 4K–16K bins, and $P \in \{2, 4\}$. As reported in the dense block of Table 1, KCP reaches 0.93, 1.01, and 1.05× the baseline throughput at 4K/P=2, 8K/P=2, and 8K/P=4, and peak memory differs by at most 0.01 GB. The only clear gain appears at 16K/P=4, where KCP reaches 1.50× the baseline throughput (1.18× by median step time). This control differs from the MoE setting in more than bin size (no MoE dispatch, TP2, one node), so it does not isolate a single factor. It does show that KCP is not a generally faster linear-attention kernel: its gains appear when full-sequence materialization becomes expensive.

6 DISCUSSION AND CONCLUSION

KCP reconciles the zigzag layout required by fused softmax-attention CP kernels with the contiguous layout required by state-passing linear-attention kernels by treating the conversion as a permutation rather than a gather. This leaves both kernels unmodified and makes the backward pass exact. On a 35B MoE hybrid, it achieves 2.5–4.7× the throughput of the all-gather baseline and trains a 256K configuration that the baseline cannot, while losses agree up to floating-point nondeterminism. Appendix D discusses the limitations of our evaluation.

AI USE STATEMENT

Large language models (LLMs) were used to assist with polishing the writing of this paper. The authors take full responsibility for its content.

REFERENCES

Y. Bai, X. Lv, J. Zhang, H. Lyu, et al. LongBench: A bilingual, multitask benchmark for long context understanding. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

W. Brandon, A. Nrusimha, K. Qian, Z. Ankner, T. Jin, Z. Song, and J. Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

T. Bu, Q. Wang, B. Zeng, H. Sun, Y. Huang, C. Cao, and J. Xu. Long-context attention benchmark: From kernel efficiency to distributed context parallelism. *arXiv preprint arXiv:2510.17896*, 2025.

Y. Chou, Z. Liu, R. Zhu, X. Wan, T. Li, C. Chu, Q. Liu, J. Wu, and Z. Ma. ZeCO: Zero communication overhead sequence parallelism for linear attention. In *Advances in Neural Information Processing Systems*, 2025.

T. Dao and A. Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning*, 2024.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

S. De, S. L. Smith, A. Fernando, A. Botev, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

Y. Du, R. Ye, et al. OpenSeeker: Democratizing frontier search agents by fully open-sourcing training data. *arXiv preprint arXiv:2603.15594*, 2026.

J. Fang and S. Zhao. USP: A unified sequence parallelism approach for long context generative AI. *arXiv preprint arXiv:2405.07719*, 2024.

R. Ghadia, M. Abraham, S. Vorobyov, and M. Ryabinin. Untied Ulysses: Memory-efficient context parallelism via headwise chunking. *arXiv preprint arXiv:2602.21196*, 2026.

P. Glorioso, Q. Anthony, Y. Tokpanov, J. Whittington, et al. Zamba: A compact 7B SSM hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *Conference on Language Modeling*, 2024.

D. Gu, P. Sun, Q. Hu, T. Huang, et al. LoongTrain: Efficient training of long-sequence LLMs with head-context parallelism. *arXiv preprint arXiv:2406.18485*, 2024.

S. A. Jacobs, M. Tanaka, C. Zhang, M. Zhang, et al. DeepSpeed Ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, 2020.

Kimi Team, Y. Zhang, Z. Lin, X. Yao, J. Hu, et al. Kimi Linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

O. Lieber, B. Lenz, H. Bata, G. Cohen, et al. Jamba: A hybrid transformer-Mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

H. Liu, M. Zaharia, and P. Abbeel. RingAttention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations*, 2024.

MiniMax. MiniMax-01: Scaling foundation models with lightning attention. *arXiv preprint arXiv:2501.08313*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

L. Ren, Y. Liu, Y. Lu, Y. Shen, C. Liang, and W. Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling. In *International Conference on Learning Representations*, 2025.

M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

W. Sun, D. Lan, Y. Zhong, X. Qu, and Y. Cheng. LASP-2: Rethinking sequence parallelism for linear attention and its hybrid. *arXiv preprint arXiv:2502.07563*, 2025a.

W. Sun, D. Qin, W. Sun, R. Zhang, et al. Linear attention sequence parallelism. *Transactions on Machine Learning Research*, 2025b.

Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, et al. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

S. Yang and Y. Zhang. FLA: A Triton-based library for hardware-efficient implementations of linear attention mechanism. https://github.com/fla-org/flash-linear-attention, 2024.

S. Yang, J. Kautz, and A. Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *International Conference on Learning Representations*, 2025.

A PLAN CONSTRUCTION DETAILS

A.1 GENERALIZED ZIGZAG SPLIT

The trailing padding pseudo-sequence requires a slight generalization of the standard zigzag split. For every packed segment i, including the padding tail, define

$$c_i = \left\lfloor \frac{\ell_i}{2P} \right\rfloor, \qquad c'_i = \frac{\ell_i}{P} - c_i. \tag{3}$$

The segment is divided into a first group of P chunks of size c_i and a second group of P chunks of size c'_i. Rank r owns chunk r in the first group and chunk $P-1-r$ in the second group. For a real sequence, $2P$ divides ℓ_i, so $c'_i = c_i$ and this reduces to the standard zigzag layout. For the padding tail, only $P \mid \ell_i$ is guaranteed, and the two chunk sizes may differ. Nevertheless, $P(c_i + c'_i) = \ell_i$, so the rule assigns every token position exactly once.

A.2 PERMUTATION FORMULATION

Applying the plan. For analysis, we stack the rank-local shards in rank order and write the distributed input as $x^{\mathrm{zz}} \in \mathbb{R}^{T \times d}$. Let $\Pi_r^{\mathrm{pack}} \in \{0,1\}^{p \times p}$ be the permutation matrix that groups rank r's local tokens by destination rank and global position, and let $\Pi_r^{\mathrm{place}} \in \{0,1\}^{p \times p}$ place its received tokens into contiguous order, where $p = T/P$.

Their distributed forms are the block-diagonal permutation matrices

$$\Pi^{\mathrm{pack}} = \mathrm{diag}\big(\Pi_0^{\mathrm{pack}}, \ldots, \Pi_{P-1}^{\mathrm{pack}}\big), \qquad \Pi^{\mathrm{place}} = \mathrm{diag}\big(\Pi_0^{\mathrm{place}}, \ldots, \Pi_{P-1}^{\mathrm{place}}\big).$$

Let $N \in \mathbb{N}^{P \times P}$ be the communication-count matrix with $N_{rq} = n_{r \to q}$, and let $A_N \in \{0,1\}^{T \times T}$ denote the permutation induced by `all_to_all_single` with these split sizes. The complete zigzag-to-contiguous redistribution is then

$$x^{\mathrm{ctg}} = \mathcal{P}_{\mathrm{zz} \to \mathrm{ctg}}\big(x^{\mathrm{zz}}\big) = \Pi^{\mathrm{place}} A_N \Pi^{\mathrm{pack}} x^{\mathrm{zz}}. \tag{4}$$

All three factors act only on the token dimension and are permutation matrices. Therefore, the reverse redistribution is

$$\mathcal{P}_{\mathrm{ctg} \to \mathrm{zz}} = \mathcal{P}_{\mathrm{zz} \to \mathrm{ctg}}^{-1} = \big(\Pi^{\mathrm{pack}}\big)^\top A_N^\top \big(\Pi^{\mathrm{place}}\big)^\top. \tag{5}$$

Here $A_N^\top$ is implemented by reversing the all-to-all communication, equivalently by transposing N and swapping the send and receive split vectors. The permutation matrices are mathematical notation only and are never materialized as dense matrices. Instead, each rank independently constructs its local index and split-size vectors from `cu_seqlens` on the CPU. For a batch containing m packed segments, plan construction takes $O(mP)$ work per rank, or $O(P)$ per segment, and requires no device memory access. The resulting plan is cached in `packed_seq_params` and reused by every linear-attention layer. In the largest configuration evaluated in this work, the cached plan occupies less than 1 MB per rank.

Algorithm 1 gives the construction of Section 3.2 in full; its outputs are the objects of that section in index form. INSPLITS and OUTSPLITS are the split-size vectors $\mathbf{n}_r^{\mathrm{send}}$ and $\mathbf{n}_r^{\mathrm{recv}}$, gathering with SENDIDX applies Π_r^{pack}, and gathering with GATHERIDX applies Π_r^{place}. This appendix works it through at $P{=}4$ with two real sequences and a padding tail; Figure 2 draws it. Let the padded lengths be $\ell_0 = 32$, $\ell_1 = 16$, and a padding pseudo-sequence $\ell_2 = 12$, so $T = 60$ and each rank holds 15 tokens.

For the real sequences $2P \mid \ell_i$, so Equation (3) gives the standard zigzag: $c_0 = 32/8 = 4$ with $c'_0 = 32/4 - 4 = 4$, and $c_1 = 2$ with $c'_1 = 2$. Rank r therefore owns chunks $\{r, 7-r\}$ of each, each of equal size.

The padding sequence is where the generalized split earns its place: $\ell_2 = 12$ is divisible by $P{=}4$ but not by $2P{=}8$, so $c_2 = \lfloor 12/8 \rfloor = 1$ while $c'_2 = 12/4 - 1 = 2$. The two halves are *different sizes*, which the textbook zigzag formula cannot express. Rank r takes one token at offset $g_2 + r$ and two tokens at $g_2 + 4 + 2(3-r)$. Every token is still claimed exactly once and every rank still receives $12/4 = 3$ of them, so the map remains a bijection with matched all-to-all split sizes — which is all correctness requires, since these tokens are loss-masked and their values are identical.

Algorithm 1 Redistribution plan at rank r (CPU, once per batch), implementing Section 3.2. $\textsc{Segs}(s)$ denotes rank s's zigzag holding: for each sequence i with $\ell = g_{i+1} - g_i$ and c, c' from Equation (3), the two triples (global start, length, local start) $(g_i{+}sc,\ c,\ g_i/P)$ and $(g_i{+}Pc{+}(P{-}1{-}s)c',\ c',\ g_i/P{+}c)$.

Require: offsets $g_0 < \cdots < g_m = T$, CP degree P, rank r

$p \leftarrow T/P$ ▷ contiguous span per rank = local zigzag length

Send side

$L \leftarrow$ empty list

for all $(g, n, l) \in \textsc{Segs}(r)$ **do**

  cut $[g, g{+}n)$ at multiples of p; append each piece $(\lfloor g'/p \rfloor,\ g',\ l{+}g'{-}g,\ n')$ to L

end for

sort L by (destination, global start)

$\textsc{SendIdx} \leftarrow$ concatenation of $[l', l'{+}n')$ over L

$\textsc{InSplits}[q] \leftarrow \sum\{\, n' : (q, g', l', n') \in L \,\}$

Receive side

$pos \leftarrow 0$; $\textsc{OutSplits} \leftarrow \mathbf{0}^P$

for $s \leftarrow 0$ **to** $P - 1$ **do**

  $O \leftarrow$ intersections of $\textsc{Segs}(s)$ with $[rp, (r{+}1)p)$, sorted by global start

  for all $[a, b) \in O$ **do**

    $\textsc{GatherIdx}[a{-}rp : b{-}rp] \leftarrow [pos,\ pos{+}b{-}a)$

    $\textsc{OutSplits}[s] \leftarrow \textsc{OutSplits}[s] + (b{-}a)$; $pos \leftarrow pos + (b{-}a)$

  end for

end for

return $\textsc{SendIdx}$, $\textsc{InSplits}$, $\textsc{OutSplits}$, $\textsc{GatherIdx}$

The reverse direction swaps the two split lists and inverts the two index vectors.

Had we instead used $c_2' = c_2 = 1$, ranks would claim only 8 of the 12 padding tokens and the send and receive split sizes would disagree, which surfaces as a shape mismatch inside `all_to_all_single` rather than as a wrong number — the failure is loud, but only if the padding case is exercised. Our CPU unit tests exercise it explicitly.

Reading Figure 2. The figure traces the plan on the worked example above ($P{=}4$: padded lengths $32{+}16$ plus a 12-token padding tail, $T{=}60$, 15 tokens per rank). Read the panels in numbered order; they show the forward trip into one linear-attention layer, with the two ends of the collective side by side. *(1)* The packed bin in global order, painted by the zigzag split: rank r owns chunks $\{r,\ 2P{-}1{-}r\}$ of every sequence (numbered), and the padding tail, divisible by P but not $2P$, splits by Equation (3) into *unequal* halves ($c_2{=}1$, $c_2'{=}2$). Contiguous span edges (dashed) ignore sequence boundaries and clip the three chunks they cross. *(2)* The same tokens device-major — what each rank actually holds: color is contiguous, but the chunk indices interleave (0 7 0 7), the order the redistribution must undo. Rank 1's holdings (outlined) are shaded by *destination*, light to dark — the partition the plan computes. *(3)* Rank 1's send buffer: those pieces sorted by (destination, global position), whose group sizes form $\mathbf{n}_1^{\text{send}}$. One `all_to_all` exchanges the buffers. *(4)* Its receive buffer: source-major arrivals, colored by sender; the receiver derives $\mathbf{n}_1^{\text{recv}}$ from the same `cu_seqlens`, so no metadata is exchanged. *(5)* The placement permutation Π^{place} sorts the arrivals into the contiguous layout the kernel consumes. Rank 1's span, listed by source underneath, is the exact dual of its zigzag block directly above (the same fifteen tokens, positions $[15, 30)$): there color was contiguous and the indices interleaved; here the indices are contiguous and the colors interleave. Afterwards, the inverse permutation restores zigzag. The piece $[24, 28)$ never travels: outlined throughout, it keeps its destination shade.

B CORRECTNESS AND DETAILED COST ACCOUNTING

B.1 CORRECTNESS PROOF

In this section, we establish that the plan of Section 3.2 defines a bijection, that the independently constructed send and receive plans agree, and that the backward pass is exactly the inverse redistribution.

Coverage. For packed segment i, Equation (3) gives

$$c_i + c'_i = \frac{\ell_i}{P}.$$

Rank r owns one disjoint chunk of each size, so it owns exactly ℓ_i/P global token positions from that segment. Across all ranks, the two chunk groups contain $P(c_i + c'_i) = \ell_i$ positions and partition the entire segment. Summing over the m packed segments gives

$$\sum_{i=0}^{m-1} (c_i + c'_i) = \frac{1}{P} \sum_{i=0}^{m-1} \ell_i = \frac{T}{P} = p$$

positions per rank. Thus, every global token position in $[0, T)$ belongs to exactly one zigzag ownership set $\mathcal{Z}_r$, including positions in the trailing padding segment when $c_i \neq c'_i$. Likewise, every global token position belongs to exactly one contiguous span I_q. The two layouts therefore partition the same set of token positions and differ only in rank ownership.

Matched counts and order. Fix a source rank r and a destination rank q, and let $\mathcal{E}_{rq}$ be the non-empty interval components of $\mathcal{Z}_r \cap I_q$. These intervals are disjoint, and their total length is

$$n_{r\to q} = \sum_{E\in\mathcal{E}_{rq}} |E| = \left(\mathbf{n}_r^{\text{send}}\right)_q = \left(\mathbf{n}_q^{\text{recv}}\right)_r. \tag{6}$$

The sender and receiver both reconstruct $\mathcal{E}_{rq}$ from the same `cu_seqlens` and order it by global start. They therefore agree on both the number and order of tokens in every $r \to q$ message without exchanging plan metadata. Consequently, `all_to_all_single` receives matched split sizes, and A_N maps every packed send-buffer position to exactly one receive-buffer position.

Exact inverse and backward. Coverage and order agreement make Π^{pack}, Π^{place}, and A_N permutation matrices. Their product in Equation (4) is therefore also a permutation, and Equation (5) gives

$$\mathcal{P}_{\text{ctg}\to\text{zz}}\big(\mathcal{P}_{\text{zz}\to\text{ctg}}(x)\big) = x. \tag{7}$$

The round trip is exact because the redistribution changes only token locations and performs no floating-point arithmetic. For a loss $\mathcal{L}$, the Jacobian is the same permutation applied independently to each hidden feature, so

$$\nabla_{x^{\text{zz}}}\mathcal{L} = \mathcal{P}_{\text{ctg}\to\text{zz}}\big(\nabla_{x^{\text{ctg}}}\mathcal{L}\big). \tag{8}$$

The backward pass therefore requires neither a reduction nor numerical scaling.

Together, the three properties above establish the correctness of the KCP redistribution. Coverage shows that every global token position is assigned to exactly one rank; matched counts and order ensure that the independently constructed send and receive plans describe the same cross-rank messages; and the inverse-permutation result proves exact round-trip recovery and the corresponding backward rule. Thus, KCP changes only token ownership and local storage order: it neither drops nor duplicates activations, and its backward pass requires no reduction or numerical scaling. Section 5.2 verifies these plan-level identities and the resulting end-to-end layer composition experimentally.

B.2 DETAILED COST ACCOUNTING

Having established the correctness of the redistribution, we now quantify how replacing full-sequence materialization with two sharded redistributions changes communication, linear-attention compute, and activation memory. The analysis considers one linear-attention layer and one rank, with $p = T/P$ local tokens, hidden size d, and b bytes per element.

Table 3: Per-layer, per-rank logical receive volume of one forward collective (d=2048, `bf16`, P=16). The per-collective ratio is P; the end-to-end communication ratio is $P/2$ because KCP crosses the layout boundary twice.

Bin T (tokens)	AllGather output	KCP output	Ratio
65,536	268 MB	17 MB	16×
262,144	1,074 MB	67 MB	16×
1,048,576	4,295 MB	268 MB	16×

One KCP collective handles $P\times$ less logical payload. In the forward pass, the gather baseline materializes a $T \times d$ receive buffer on every rank, for a logical receive volume of Tdb bytes. One KCP all-to-all instead returns a $p \times d$ buffer, or $(T/P)db$ bytes. The per-collective ratio is therefore exactly P. Table 3 instantiates these volumes: at 256K tokens, one gather materializes just over 1 GB per layer per rank, whereas one KCP all-to-all returns 67 MB.

These figures are logical collective volumes: for each collective, we count the per-rank tensor it materializes (all-gather), consumes (reduce-scatter), or redistributes (all-to-all), including any local or self-directed portion. Actual bytes carried by the network depend on the communication algorithm, self-transfers, and topology; logical volume provides the implementation-independent comparison used throughout this section.

End-to-end communication improves by $P/2$, not P. The comparison changes once all collectives in a training step are counted. In the forward pass, the baseline performs one all-gather, whereas KCP performs two all-to-alls, one at each layout boundary. Their backward operators have the same counts: one reduce-scatter for the baseline and two reverse all-to-alls for KCP. Without activation recomputation, the resulting logical volumes are $2Tdb$ and $4pdb$, respectively.

Full activation recomputation adds one more forward evaluation during backward. It therefore adds one gather to the baseline and two all-to-alls to KCP, giving

$$V_{\text{base}} = 3Tdb, \qquad V_{\text{KCP}} = 6pdb = 6\frac{T}{P}db, \qquad \frac{V_{\text{base}}}{V_{\text{KCP}}} = \frac{P}{2}. \tag{9}$$

The ratio is $8\times$ at P=16 and remains $P/2$ when recomputation is disabled. This is a communication-volume ratio, not a predicted wall-clock speedup; quoting the per-collective $P\times$ ratio as either would overstate the result.

Linear-attention compute and activation memory improve by $P\times$. For fixed feature dimensions, the sequence-dependent compute and stored activations of a linear-attention kernel scale linearly with the number of tokens evaluated on each rank. The baseline evaluates all T tokens per rank, whereas KCP evaluates only $p = T/P$. Thus the corresponding kernel terms change as

$$C_{\text{base}} = \Theta(T), \quad C_{\text{KCP}} = \Theta(T/P), \qquad M_{\text{base}} = \Theta(Td), \quad M_{\text{KCP}} = \Theta((T/P)d).$$

This is a $P\times$ reduction in the linear-attention terms, not a claim of $P\times$ end-to-end training speedup: full-attention layers, MoE blocks, and the layout bridge remain. The peak-memory difference also includes the transient collective buffer: the baseline materializes $T \times d$, while KCP remains at $(T/P) \times d$. These predictions motivate the throughput and memory measurements in Section 5.2.

C ADDITIONAL EXPERIMENTAL DETAILS

C.1 EXPERIMENTAL SETUP DETAILS

The target models used in the experiments are summarized below. Both use a 1:3 full:linear attention ratio.

Table 4: Target models. Both use a 1:3 full:linear attention ratio.

Model	Layers	Full/Linear	d	Experts	Type
Qwen3.5-9B	32	8 / 24	4096	–	dense
Qwen3.6-35B-A3B	40	10 / 30	2048	256	MoE

C.2 TOKEN-BUDGET CONTROL

Controlling the token budget. Bin size, per-GPU budget and CP degree are not independent: $T = (\text{tokens per GPU}) \times P$. Sweeping P at a fixed per-GPU budget therefore sweeps T too, so we instead declare T and derive the budget as T/P. A second confound is specific to packed training: the packer places an oversized row alone in its own bin, whose total then *exceeds* the nominal budget, so a bin target only means anything if essentially every row fits. In our corpus all but a handful of rows fit a 256K bin — the exceptions are the few that exceed 262K, the tail reaching 368K — and 88% fit a 128K bin, but only 17% fit a 32K bin — an unbucketed small-bin sweep would measure the overflow fraction rather than the bin size. We therefore bucket by length and draw each configuration from a bucket whose rows fit (Table 5), so the declared T is the bin size actually trained rather than an average over overflowing bins. Even the smallest bucket supports far more steps than any configuration here consumes.

Table 5: Length buckets used to keep T a controlled variable.

Bucket (tokens)	Rows	Tokens	Steps at bucket midpoint
16K–24K	863	17.7M	∼860
24K–41K	1,798	59.3M	∼1,800
41K–74K	3,389	192.7M	∼3,360
74K–147K	4,109	427.1M	∼3,860
147K–262K	523	85.8M	∼420

C.3 ADDITIONAL VIEWS OF THE MAIN RESULTS

Table 6 summarizes three representative settings from Table 1.

Table 6: **Representative settings.** Within each block the model, data, optimizer, and parallel layout are identical; only the linear-attention CP strategy differs. "Comm/rank" is the per-rank receive volume of one forward collective at one linear-attention layer (Proposition 2). ALLGATHER is the strategy deployed in the Megatron-based stack we build on (Shoeybi et al., 2019); alternatives that constrain head counts or require kernel changes are discussed in Section 2. Full sweep: Table 1.

Setting	Method	Comm/rank	Trains	Tok/s/GPU	Peak GB	Speedup
35B MoE, T=128K, P=16	ALLGATHER	T	✓	189	44.9	1.00×
32×H100	KCP	$T/16$	✓	**881**	**36.6**	**4.66×**
35B MoE, T=256K, P=16	ALLGATHER	T	OOM	—	—	—
32×H100	KCP	$T/16$	✓	**484**	**53.3**	**only arm**
9B dense, T=16K, P=4	ALLGATHER	T	✓	837	54.2	1.00×
8×H100	KCP	$T/4$	✓	**1,255**	54.2	**1.50×**

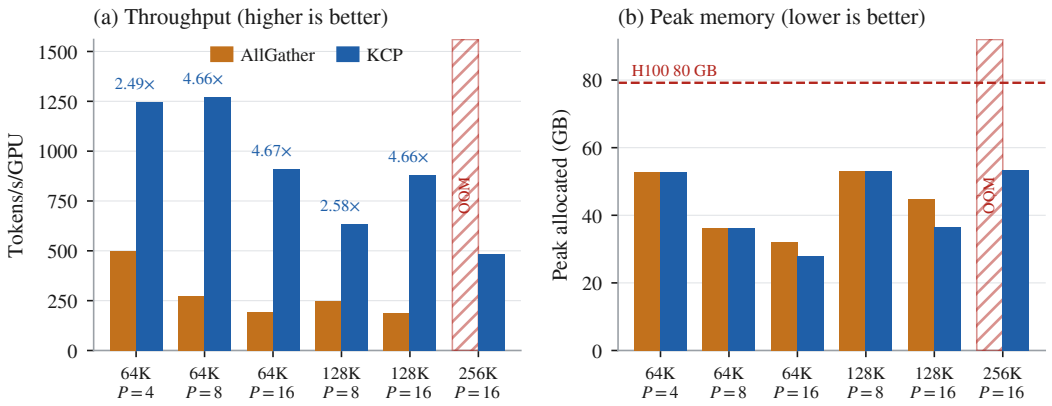

Figure 6: The sweep of Table 1 drawn as bars. KCP is faster at every configuration both arms can run, and is the only arm that runs at 256K/P=16. Hatched bars are out-of-memory failures, drawn rather than omitted; they span the axis because no value was measured, not because the value is large.

C.4 Step-time statistics

Per-step training times are bimodal on both arms: most steps take a roughly constant time, but some take several times longer. Figure 3 shows the 128K/P=16 pair. After the five warmup steps, the baseline alternates between about 20 and 98 s/step through the final step, whereas KCP stays near 10 s/step except for two long steps, which remain in the aggregate. Table 7 reports, for every paired configuration, the mean and median of the 25 post-warmup step times and the number of long steps, defined as steps that take more than twice the fastest post-warmup step of the same run. Because both arms process identical tokens at every step, the ratio of mean step times equals the ratio of aggregate throughputs in Table 1. The baseline has more long steps at every MoE configuration, and KCP is faster at every MoE configuration under either statistic. The two statistics can differ substantially, because the median reports whichever mode holds the majority of steps. The long steps are reproducible rather than random: a repeat of the 256K/P=16 KCP run (Appendix C.6) matches the original's step time to within 10% at every one of the 30 steps, long steps included.

Table 7: Post-warmup step-time statistics (seconds, 25 steps per run). "Long" counts steps slower than twice the fastest post-warmup step of the same run. The mean ratio equals the throughput ratio of Table 1; the median ratio is the ratio of median step times.

		AllGather			KCP			AllGather / KCP	
Model	Config (T/P)	Mean	Median	Long	Mean	Median	Long	Mean	Median
35B MoE	64K / 4	74.2	48.9	19	29.7	14.4	4	2.49×	3.40×
	64K / 8	65.2	45.4	18	14.0	9.8	1	4.66×	4.64×
	64K / 16	44.4	15.2	9	9.5	8.4	1	4.67×	1.81×
	128K / 8	127.8	106.9	24	49.6	14.6	10	2.58×	7.31×
	128K / 16	85.8	97.7	16	18.4	10.8	2	4.66×	9.08×
	256K / 16		OOM		38.2	13.8	7	—	—
9B dense	4K / 2	7.8	6.3	1	8.4	6.2	1	0.93×	1.01×
	8K / 2	6.9	6.8	0	6.8	6.7	0	1.01×	1.02×
	8K / 4	8.8	8.2	1	8.3	8.4	0	1.05×	0.98×
	16K / 4	12.8	10.0	3	8.5	8.5	0	1.50×	1.18×

C.5 Bridge profiling

The layout bridge accounts for 1.8–2.7% of GPU-busy time on the profiled ranks. We profile the 35B MoE model at 256K/P=16 on 32 H100 GPUs. Because the bridge and MoE dispatch share the same `all_to_all` annotation, we distinguish them by tensor rank: the bridge moves $[T/P, 1, d]$, whereas dispatch moves [tokens, d]. Figure 7 and Table 8 report the GPU-time breakdown

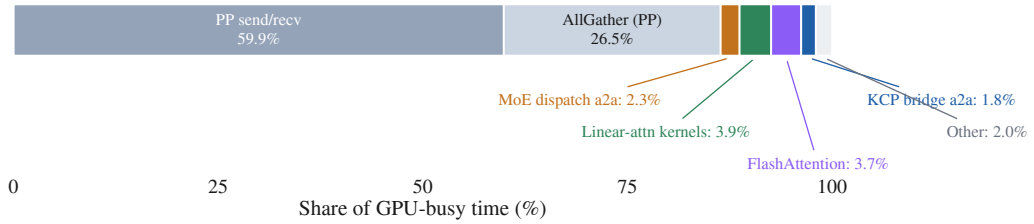

Figure 7: GPU-busy time on rank 0 at 256K/P=16 over two profiled rollouts (97.2 s total). The KCP bridge accounts for 1.8%, less than the MoE dispatch all-to-all and less than half the linear-attention kernel time.

Table 8: Per-component GPU time behind Figure 7 (rank 0, 256K, P=16, 35B MoE, 32×H100; 97.2 s of GPU-busy time over two profiled rollouts).

Component	Time (ms)	% of GPU busy
Pipeline send/recv (NCCL)	58,222	59.9
All-gather (pipeline-related, NCCL)	25,760	26.5
Linear-attention kernels (+ causal conv)	3,757	3.9
FlashAttention (full-attention layers)	3,568	3.7
MoE dispatch all-to-all	2,243	2.3
KCP bridge all-to-all	**1,736**	**1.8**
Everything else	1,930	2.0

Table 9: System robustness checks at 256K on the 35B MoE model and 32×H100. Each row is a separate run; the default row repeats the 256K/P=16 configuration of Table 1. The CP32 throughput is not directly comparable to the CP16 reference because the pipeline configuration also changes.

Configuration	Outcome	Tokens/s/GPU
KCP, default (TP1·PP2·CP16·EP8)	trains	483
KCP, optimized MoE dispatch	trains	491
KCP, TP1·PP1·CP32	trains	915

on rank 0 over two profiled rollouts: 97.2 s of GPU-busy time, of which the KCP bridge accounts for 1.8%. Across six sampled ranks spanning both pipeline stages (ranks 0, 1, 8, 16, 24, and 31), the bridge's share ranges from 1.8% to 2.7% (median 2.1%). NCCL send/receive and all-gather kernels outside the bridge account for 86% of rank 0's GPU-busy time, and this time includes waiting on peers. Relative to computation, the bridge's 1.7 s is therefore more substantial than its percentage suggests: it is 46% of the time spent in linear-attention kernels and 24% of the time spent in linear- and full-attention kernels combined, although it remains below the MoE dispatch all-to-all. Plan construction runs once per batch on the CPU and is amortized across the 30 linear-attention layers. Within the bridge, `all_to_all` dominates; the `index_select` gathers that pack and place tokens take only 79 ms of CPU time in total. The measured overhead is therefore dominated by communication.

C.6 Dispatcher and parallelism checks

The 256K result is robust to the tested dispatcher and parallelism choices. Table 9 tests whether the 256K result depends on the MoE dispatcher or the default TP/PP split. The default row is a separate run of the configuration in Table 1; its throughput (483 tokens/s/GPU) matches the original run (484) to within 0.2%. Enabling the optimized dispatcher changes throughput by 1.7% (483 to 491 tokens/s/GPU), and the run remains feasible. Repartitioning the same 32 GPUs as TP1·PP1·CP32 also trains, at 915 tokens/s/GPU and 56.2 GB peak memory. This throughput is not directly comparable because removing pipeline parallelism also removes its send/receive traffic; the result is used only as a feasibility check.

D LIMITATIONS

Our evaluation covers one architecture family, Gated DeltaNet hybrids with one full-attention layer in four, in dense and MoE variants, on up to 32 H100 GPUs; we make no claims beyond 256K-token bins. Each configuration is a single 30-step run. KCP moves the memory wall rather than removing it: the terms it shards still grow with T/P, and at small bins it brings no speedup. It also requires linear-attention kernels with native contiguous-layout CP support. Finally, we compare against the deployed all-gather strategy; an empirical comparison with head sharding and zigzag-aware state passing (Section 2) remains future work.