FLATTENING EVERY MEMORY PEAK IN LONG-CONTEXT MIXTURE-OF-EXPERTS TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Training a Mixture-of-Experts (MoE) model at long context or large batch size fails as soon as any one component's peak allocation exceeds device memory, so the target is every peak at once, not the average footprint. Four are left unbounded by the parallelism plans in common use, and each grows differently: expert dispatch with the routing matrix, the vocabulary projection with tokens times vocabulary, gradient checkpoint boundaries with depth times sequence length, and optimizer state with parameter count. Which one runs out first changes with the model, the context length, and the device count, so lowering the largest only exposes the next. We bound all four with schedules whose GPU working set is fixed at launch: PipelinedLLEP extends least-loaded expert parallelism with a cap on the tokens each source contributes to a dispatch chunk, Ring-DTP circulates activations or weight shards around a ring at the vocabulary projection and folds each block of logits into an online log-sum-exp, Selective checkpoint offload (SCO) keeps the one long-lived tensor of each checkpoint boundary in CPU memory, and OffloadStreamAdamW turns the serial CPU Adam update of optimizer offload into a bucket pipeline. All four change only the order and granularity of computation and data movement, so the loss and gradients stay exact. In matched component tests, they cut the MoE dispatch peak by up to 59.3% without losing throughput and the vocabulary projection peak by 86.6%, and speed the offloaded optimizer step by $2.05\times$. Composed on MoE models from 120B to 667B parameters, they train at 1M context length, 8–$32\times$ the reach of a tuned FSDP2 baseline, and up to $10.4\times$ its throughput.

1 INTRODUCTION

Mixture-of-Experts (MoE) models are often trained at long context or with large batches at scale. Because adding GPUs is rarely an available remedy, memory is traded for time. Every standard trade charges time: gradient checkpointing recomputes (Chen et al., 2016; Korthikanti et al., 2023), state sharding communicates (Rajbhandari et al., 2020; Zhao et al., 2023), and optimizer offload serializes the update on the host (Ren et al., 2021). A training step dies when any component's peak memory occupancy exceeds the device's capacity. Reaching longer context, larger batches, or larger models means holding every component under the device ceiling at the same time.

Which peak runs out first depends on the configuration. Peak memory is a maximum over *live sets*, the groups of tensors that must be resident simultaneously. In this work, we explore four components that are left unbounded by the parallelism plans in common use: expert dispatch grows with the routing matrix the router draws (Lepikhin et al., 2020; Rajbhandari et al., 2022), the vocabulary projection with the token-vocabulary product (Wijmans et al., 2025), retained gradient checkpointing boundaries with depth times sequence length (Chen et al., 2016), and AdamW state with parameter count (Loshchilov and Hutter, 2019). Because these grow at different rates, the largest of the four depends on the configuration: logits dominate at large vocabulary and long context, MoE dispatch at high routing imbalance, checkpointing boundaries at high depth, and optimizer state at large parameter count on few devices.

Each operator can be enabled separately, so a configuration incurs only its costs. All four change only the order and granularity of computation and data movement, leaving the model, parameterization,

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.

optimizer, precision, and loss untouched: no low-rank adapters, quantized state (Dettmers et al., 2022), or approximate routing or attention. Memory and throughput figures are therefore directly comparable with standard full-parameter BF16 training.

Four bounded-streaming operators. (*i*) Expert parallelism shards experts across ranks (GPUs), so a rank must hold every token routed to the experts it owns, and the router decides how many that is (Lepikhin et al., 2020; Rajbhandari et al., 2022). Least-loaded expert parallelism (LLEP) removes the time cost of routing skew by moving routed work to idle ranks without changing any token's expert choice (Nguyen et al., 2026), but a rank still allocates for its whole routed batch at once. Our proposed PipelinedLLEP delivers the same routes in chunks and caps the tokens any source may place in a chunk, so a receiver's buffers follow that cap and routing skew lengthens the schedule instead of enlarging the buffers.

(*ii*) The vocabulary projection materializes a tokens-by-vocabulary logit tensor, which dominates memory once both factors are large. Fused cross-entropy kernels avoid it on a single device by streaming the log-sum-exp over vocabulary blocks (Wijmans et al., 2025; Hsu et al., 2024), and Megatron shards the weight but requires every rank to hold the same batch (Shoeybi et al., 2019), reducing effective batch size. Our Ring-DTP (ring data-tensor-parallel vocabulary projection) lets each rank keep its own distinct batch and circulates either the activations or the weight shards around a ring until every batch has met every shard, the exact loss and gradients are computed yet the large tensor is never formed.

(*iii*) Gradient checkpointing buys memory with recomputation, yet one tensor per checkpointed layer, its input, still lives from the forward pass until that layer is recomputed (Chen et al., 2016). Our SCO (selective checkpoint offload) offloads a host-budget subset of those inputs into host memory, and prefetches them back one layer before the layer is recomputed.

(*iv*) Optimizer offload stores AdamW state on CPU RAM, but a CPU update operation is slow, during which the GPU is idle and, having released its activations, nearly empty (Ren et al., 2021). Instead, our OffloadStreamAdamW (streamed offloaded AdamW) streams the parameters from host RAM to GPUs, computes the weight update there, and writes it back. It overlaps communication and computation.

We compose the four operators in a hybrid module-specific parallelism regime. End to end on MoE models of 120B, 241B, and 667B parameters, the full stack trains at one-million-token context, 8–32$\times$ the reach of a tuned FSDP2 baseline, at up to $10.4\times$ its throughput and up to $12\times$ its largest global batch (§4).

2 Four Scaling Axes

Let Θ be the number of trainable parameters, W the number of devices, N the tokens resident per rank, H the hidden width, V the vocabulary size, and b the bytes per activation scalar. AdamW (Loshchilov and Hutter, 2019) ordinarily requires 16Θ bytes for working weights, gradients, master weights, and moments. ZeRO-3 shards this persistent state, sequence parallelism reduces attention activations, and expert parallelism partitions expert weights (Rajbhandari et al., 2020; Zhao et al., 2023; Jacobs et al., 2023; Rajbhandari et al., 2022). These methods leave the four live sets in Table 1 unbounded: their sizes follow workload and model dimensions rather than the sharding degrees.

Common requirements. The four rows scale differently and need different schedules. They all preserve the forward values and gradients of the implementations they replace. They may only change the order of computation but the results must be the same. Their memory bounds also hold during checkpoint recomputation, so they retain nothing sized by the routing matrix at the step's memory peak. Together, these bounds form the per-rank budget in Equation (6); established components handle the remaining device memory.

Table 1: The four live sets of tensors that this paper targets to reduce peak memory. For MoE, R_d counts routes received by rank d; $q = kc_{\text{eff}}$ is the per-source route budget; E_p and P are the expert- and vocabulary-tensor-parallel degrees; For SCO, $\mathcal{S}$ is the offloaded-boundary set; and $N_{\max}$ and β are the token and optimizer-bucket limits.

Component	Unbounded term	Stream axis	Bounded GPU set
Expert dispatch	$2R_dHb$; routing matrix	Token chunks	$2E_pqHb$
Vocabulary logits	$\mathcal{O}(NVb)$; $N \times V$	Vocabulary blocks	$\mathcal{O}(NVb/P)$
Checkpoint boundaries	$\sum_{\ell \in \mathcal{S}} N_\ell Hb$; depth $\times$ length	Decoder layers	$\leq 2N_{\max}Hb$ in HBM
Offloaded AdamW	Serial $12\Theta/W$-byte host update	Parameter buckets	$\mathcal{O}(\beta)$ HBM staging

3 Four Bounded-Streaming Operators

Each subsection gives the mechanism, the bound it delivers, and a benchmark that isolates the live set the operator governs. Benchmarks run on one eight-H200 node and hold the input fixed across the systems compared; configurations are in §A. End-to-end results follow in §4.

3.1 PipelinedLLEP: Receiver-Bounded Expert Dispatch

Under expert parallelism each rank owns only some of the experts, so a token routed to one of them has to travel to the rank that owns it, its *destination* d. The number of routes R_d to d determines the size of its dispatch buffers (Lepikhin et al., 2020; Rajbhandari et al., 2022). That count grows with three things: the batch size, the routing top-k k, and routing imbalance. LLEP removes the third of these by moving a popular expert's overflow onto ranks that would otherwise be idle (Nguyen et al., 2026). The first two remain: LLEP still holds the rank's entire routed batch as the all-to-all materializes that batch in one tensor (Figure 2).

PipelinedLLEP instead splits the batch into chunks, using a maximum token budget c, and processes the chunks in a pipeline that overlaps communication with computation. The budget applies to each sending rank separately: a rank puts at most c of its own tokens into any one chunk. The number of chunks K is computed as:

$$K = \min(\lceil N/c \rceil, K_{\max}), \qquad c_{\text{eff}} = \lceil N/K \rceil, \qquad R_d^{(i)} \leq E_p k c_{\text{eff}}, \tag{1}$$

where N is the number of tokens on a rank, k is the router's top-k, E_p is the expert-parallel degree, and $K_{\max}$ is a predetermined limit on the number of chunks. K is the number of chunks, c_{eff} is how many tokens a rank actually puts in one chunk, and $R_d^{(i)}$ is the number of routed tokens that destination d receives in chunk i. Each of the E_p ranks puts at most c_{eff} tokens into chunk i, and each token goes to k experts, so one rank sends at most kc_{eff} routes in that chunk no matter which experts its tokens chose. Summing over the E_p senders produces $E_p k c_{\text{eff}}$. No routed token is dropped to make this hold. Equation (1) contains only E_p, k, and c_{eff}, and none of them depend on what the router does (Figure 1a,b). A skewed router still sends more routes to one destination than to another, but it cannot push any buffer past this limit. When $N \leq cK_{\max}$, the chunk limit is never reached and $c_{\text{eff}} = c$, so the buffers take at most $2E_pkcHb$ bytes together at hidden width H and b bytes per value. When $N > cK_{\max}$, the number of chunks stops at $K_{\max}$ and each buffer is capped at a fixed $1/K_{\max}$. Either way the limit is known beforehand.

Why overlap alone is not enough. The K chunks run through a three-stage pipeline: dispatch, then the grouped expert matmul (Gale et al., 2023), then combine. While chunk i is on the tensor cores, the dispatch of chunk $i{+}1$ and the combine of chunk $i{-}1$ use the interconnect. Other systems build this same pipeline, but they choose the number of chunks to make the iteration fast (He et al., 2022; Shi et al., 2023; Hwang et al., 2023; Shi et al., 2024); we choose it to cap memory instead, which is a different criterion and gives a different guarantee. Overlap on its own also does not lower the peak during training, because a loop that calls the expert layer once per chunk keeps every chunk's autograd graph alive until the layer's backward pass runs. PipelinedLLEP therefore wraps each chunk's expert matmul in a reentrant gradient checkpoint, nested inside the decoder layer's non-reentrant gradient checkpoint (Chen et al., 2016; PyTorch Contributors, 2026b). The inner checkpoint frees one chunk's

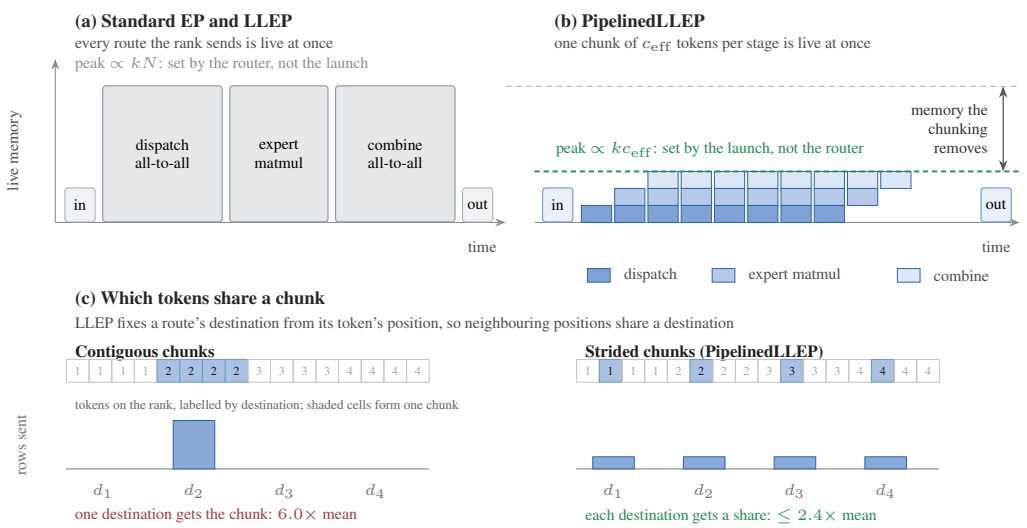


Figure 1: PipelinedLLEP caps how many tokens each rank places in a chunk. **(a)** Dispatching the whole batch at once causes peak memory spike. **(b)** Instead, with K chunks, a stage holds $1/K$ as much and the stages overlap, so peak memory stays under the limit of Equation (1). **(c)** LLEP rebalances loads by arranging tokens into contiguous indices. But when they are split into chunks, a chunk of consecutive tokens sends almost all of its routes to the same rank, reintroducing routing imbalance. PipelinedLLEP selects tokens for each chunk in strided order, spreading each chunk's routes over the destinations, which balances the load for each chunk, delivering up to $1.35\times$ in speed and up to $1.37\,\mathrm{GiB}$ of peak memory (Table 8).

intermediate tensors before the next chunk allocates its own, both forward and backward. §B.1 gives the details and the gradient accounting.

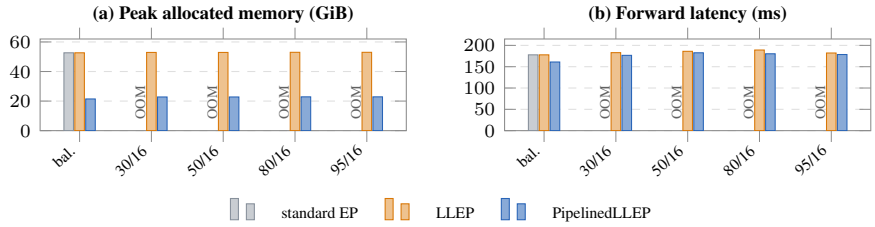
Which tokens go in which chunk. LLEP picks the rank that handles a route from the position of its token in the expert's global token list. A chunk of consecutive positions inherits that grouping, so a chunk covering the relocated tail of a popular expert sends everything it has to a single helper rank, even when the layer's plan is balanced overall. PipelinedLLEP therefore gives chunk i the positions $i, i{+}K, i{+}2K, \ldots$ (Figure 1c), which spreads each chunk's routes over the destinations. Nothing else changes: the same tokens are sent, the relocation plan is the same, and every route goes to the same destination as before. Striding runs 1.03–$1.35\times$ faster and saves up to $1.37\,\mathrm{GiB}$ of peak memory (Table 8).

Isolated benchmark. As shown in Figure 2, PipelinedLLEP saves 59.3% over LLEP at $1.10\times$ its speed; the saving grows with the unpartitioned live set. Peak memory is deterministic in the configuration and verified empirically.

3.2 Ring-DTP: Exact Vocabulary Projection over Distinct Batches

A token's cross-entropy loss does not need its whole logit row, only a running maximum, exponential sum, and the logit of its target. Those three scalars are the same size regardless of vocabulary size, so the vocabulary can be visited one block at a time and the logit tensor never has to exist. Fused cross-entropy kernels use this on a single device (Wijmans et al., 2025; Hsu et al., 2024); the difficulty here is that the batch and the projection weight are *both* sharded across ranks.

Rank r holds a distinct local batch $X_r \in \mathbb{R}^{N\times H}$ with targets t_r and owns the vocabulary interval $\mathcal{V}_r$ of size V/P, so $W_{\mathrm{voc}} \in \mathbb{R}^{H\times V}$ is column-sharded as $W_r := W_{\mathrm{voc}}[:, \mathcal{V}_r]$. Megatron's vocabulary-parallel uses the same weight partition but requires every rank to hold the same X (Shoeybi et al., 2019), which reduce the effective batch size and long-context scalability. By using a data-parallel-like layout, Ring-DTP preserves the distinct X_r and arranges P ring-like rounds of data transfer and computation. Over those rounds, every batch X_j is co-located with every shard W_r once (Figure 3).



Shape	LLEP (GiB)	PipelinedLLEP (GiB)	Peak saved	Speedup vs. LLEP
65K tokens/rank, H=7168, top-8	52.7–53.0	**21.4–22.9**	**56.9–59.3%**	1.01–1.10×

Figure 2: Expert dispatch as routing skew grows, at 65,536 tokens per rank, $H = 7168$, $I = 2048$, 128 experts, top-8, and $c = 6554$ ($K = 10$). A profile skew-level label gives the degree of routing imbalance (such as 95/16, which means 95% tokens routed to hottest 16 experts). §B.2 defines the family. At this shape standard EP runs out of memory as soon as the router is severely skewed. PipelinedLLEP saves up to 59% peak memory without significant slowdown. More details are provided in Appendix §B.2 and Tables 6 and 7.

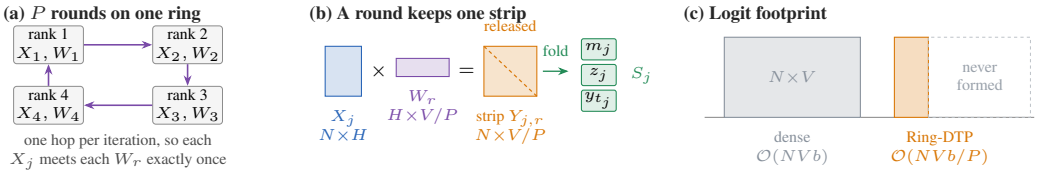


Figure 3: Ring-DTP computes exact vocabulary loss without materializing full logits. **(a)** Across P rounds, every distinct local batch meets every vocabulary shard once (shown for $P = 4$). **(b)** Each round folds one logit strip into $S_j = (m_j, z_j, y_{t_j})$ and releases it. **(c)** At most V/P columns are live, reducing logit memory by P.

A round forms only the logit strip $Y_{j,r} = X_j W_r \in \mathbb{R}^{N \times V/P}$, folds its partial normalizer and target logit into the running per-token state $S_j = (m_j, z_j, y_{t_j})$, and releases the strip. Peak logit memory is then

$$M_{\text{logit}}^{Ring-DTP} = \mathcal{O}(NVb/P), \tag{2}$$

where b is the number of bytes per value. For one token, the logits y of each strip update the running state in the standard online-softmax form (Milakov and Gimelshein, 2018; Dao et al., 2022):

$$m' = \max(m, \max_v y_v), \qquad z' = e^{m-m'} z + \sum_v e^{y_v - m'}. \tag{3}$$

After all P rounds the exact negative log-likelihood is $m + \log z - y_t$. Backward recomputes the gradients one strip at a time from the saved normalizer via the same ring, so neither pass stores a full logit tensor.

Which tensor travels, and at what price. Either the activations X_j or the weight shard W_r can move. Ring-DTP chooses the cheapest option dynamically. The move-weights schedule (Figure 9) keeps (X_j, t_j, S_j) on its rank and sends W_r around the ring, so forward needs no return hop, unlike a ring that can only move one of the two tensors (Liu et al., 2024). Bytes sent per hop are $\mathcal{O}(NH)$ if activations move and $\mathcal{O}(HV/P)$ if weights move. We move weights exactly when $N > V/P$. Backward replays the same rounds, and the gradient of the moving tensor travels with it. Sharding also cuts persistent projection storage and its backward workspace by P. The cost is $P-1$ sequential hops per pass. We therefore choose the smallest P for which the NV/P strip fits. Target-column ownership and gradient derivations are in §C.

Isolated benchmark. Table 2 covers both auto-mode branches. The standard peak scales with N, whereas Ring-DTP holds only one strip. Doubling N nearly doubles the standard baseline, from

Table 2: Vocabulary projection at $P = 8$ on a distinct local batch of N/GPU with $H = 7{,}168$ and $V = 200{,}000$ in FP32, measured over hidden states to target-token log-probs and backward. The standard reference materializes full logits and log-softmax. Auto-mode picks move-weights when $N > V/P$, which is the case for $N = 16{,}384$, while $N = 32{,}768$ switches to move-activations. Per-tensor detail is in Tables 10 and 11.

		Forward+backward peak (GiB)			Latency (ms)		
N	Schedule	Standard	Ring-DTP	Saved	Standard	Ring-DTP	Cost
16,384	move-activations	42.462	7.322	82.8%	1001.2	1052.2	+5.1%
32,768	move-weights	79.521	**10.622**	**86.6%**	2065.2	2155.8	+4.4%

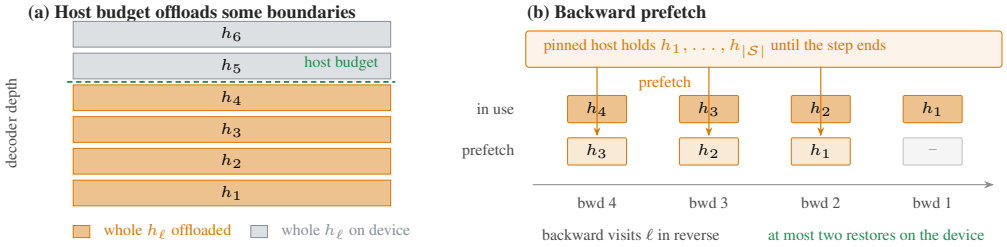


Figure 4: SCO offloads checkpoint boundaries up to a host budget and prefetches them during backward. **(a)** Each bar is one layer's checkpoint boundary h_ℓ. Orange bars fit the host budget and are copied entirely to pinned memory; gray bars stay on the device. **(b)** During backward, while h_ℓ is in use, $h_{\ell-1}$ is copied back asynchronously.

42.5 to 79.5 GiB, while Ring-DTP moves from 7.3 to 10.6 GiB, so the saving widens as context grows. At 1M context, auto-mode moves weights because its $\mathcal{O}(HV/P)$ hop payload is independent of token count.

3.3 SCO: Exact Gradient Checkpointing Boundary Offload

Under gradient checkpointing, the layer input $h_\ell \in \mathbb{R}^{N_\ell \times H}$ stays on the device from the forward pass until that layer is recomputed in backward. This retained input is the checkpoint boundary.

SCO moves a subset of those boundaries to pinned host memory. It walks the checkpointed layers in forward order and offloads each boundary that still fits in the host budget; those layers form the set $\mathcal{S}$ (Figure 4a). Each selected boundary is copied to the host asynchronously during forward, and the device copy is then freed. Layers that do not fit keep their boundaries on the device.

Backward visits layers in reverse. While layer ℓ recomputes from h_ℓ, the next boundary $h_{\ell-1}$ is copied back from the host on a separate stream (Figure 4b). At most two restored boundaries are therefore live on the device at once. Let $N_{\max}$ be the per-rank token ceiling. The restore working set and the host occupancy of $\mathcal{S}$ then satisfy

$$M_{\text{HBM}}^{SCO} \le 2N_{\max}Hb, \qquad M_{\text{host}} \le |\mathcal{S}|N_{\max}Hb. \tag{4}$$

The offloaded set stays in host memory until the step ends. Appendix §D provides more details.

Isolated benchmark. Table 3 shows the HBM–RAM tradeoff: device peak falls monotonically with the host budget, allowing more memory headroom for other GPU tasks. Throughput changes by at most 1.9%. The largest possible batch size (without OOM) rises by 35.71% (Table 13).

3.4 OffloadStreamAdamW: Bounded GPU Updates over Host CPU State

When HBM capacity is at the limit, it makes sense to offload the optimizer and its state to CPU (Ren et al., 2021; Rajbhandari et al., 2021), and the price is that a CPU AdamW update is slow, during that time the GPU device is idle and, having released its activations, largely empty in the HBM. We

Table 3: Matched SCO budget sweep on gpt-oss-20b (OpenAI, 2025), 8xH200 GPUs, sequence and expert parallelism of degree eight. Every policy processes a configured global batch size of 556432 tokens. Full sweeps are in Tables 12 and 13.

Host budget	Boundaries	Peak HBM (GiB)	Node RAM (GiB)	Throughput (tok/s/GPU)	Largest 10-step batch
Off	0 / 47	139.790	402.517	2,641	557,056
8 GiB	21 / 47	133.546	487.990	**2,692**	589,824
16 GiB	42 / 47	125.677	573.473	2,687	622,592
Full	47 / 47	**123.728**	593.410	2,687	**655,360**

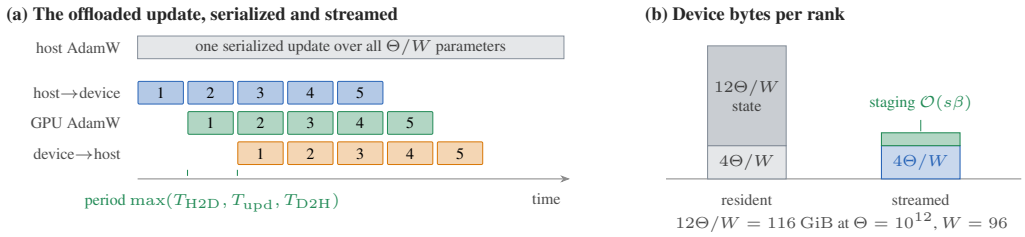


Figure 5: OffloadStreamAdamW. **(a)** Optimizer offload serializes one CPU update over all Θ/W parameters. OffloadStreamAdamW instead partitions Θ into buckets and rotates them through transfer, update, and write-back streams, leveraging communication-computation overlap. **(b)** Device bytes per rank: streaming adds only $\mathcal{O}(s\beta)$ of staging.

make use of the idle GPU as the update engine by streaming CPU-resident states to GPU and back. Specifically, OffloadStreamAdamW partitions each rank's state into buckets of at most β parameters and rotates them through s staging slots and three streams: (*i*) host-to-device transfer of master weights, moments, and final gradients; (*ii*) a fused GPU AdamW update and bf16 working-weight refresh; and (*iii*) write-back of updated fp32 state (Figure 5). For G buckets,

$$T_{\text{stream}} = G\max(T_{\text{H2D}}, T_{\text{update}}, T_{\text{D2H}}) + \mathcal{O}(T_{\text{H2D}} + T_{\text{update}} + T_{\text{D2H}}), \quad M_{\text{stage}} = \mathcal{O}(s\beta). \tag{5}$$

The floor for any offloaded optimizer is the host-link round trip of $12\Theta/W$ bytes, so the design goal is to hide the update and the write-back behind that transfer.

Isolated benchmark. Table 4 compares the OffloadStreamAdamW against the CPU Adam (Ren et al., 2021) that it replaces, with the same fp32 master weights and moments on the host. Rotating the same update through the idle GPU leads to a 2.05× speedup. Two slots are enough to keep the transfer stream busy. Making the queue deeper but increasing staging slots s leaves step time unchanged. That insensitivity to s is consistent with the pipeline being transfer-bound at the $12\Theta/W$ host-link floor of Equation (5).

3.5 Composition and the Per-Rank Budget

The four operators use a hybrid module-specific parallelism regime over the same W ranks. Overlapping groups of D, E_p, and P ranks handle Ulysses distributed attention, expert dispatch, and the vocabulary projection, while ZeRO-3 shards dense weights across all ranks (Jacobs et al., 2023; Rajbhandari et al., 2020). These group sizes need not multiply to W; §A gives the execution order. The per-rank device budget is

$$M_{\text{peak}}^{\text{integrated}} = \underbrace{\tfrac{4\Theta}{W}}_{\text{weights, grads}} + \underbrace{\mathcal{O}(s\beta)}_{\text{opt. staging}} + \underbrace{\mathcal{O}(NHb)}_{\text{attention}} + \underbrace{\mathcal{O}(E_pqHb)}_{\text{dispatch}} + \underbrace{\mathcal{O}(NVb/P)}_{\text{strips}} + \underbrace{\mathcal{O}(NHb)}_{\text{boundaries}}. \tag{6}$$

Every quantity on the right is fixed by the model or launch configuration once $N \le N_{\max}$, so the configuration can be checked for feasibility before launch.

Table 4: Optimizer step on gpt-oss-20b, 8xH200 GPUs. The baseline is the AVX-vectorized CPU Adam kernel of ZeRO-Offload. Both sides keep the same fp32 master weights and moments in host memory. Staging is resident memory per GPU. Bucket-size sweeps are in Table 14.

Configuration	Step time (s)	Speedup	Staging (GiB/GPU)
CPU Adam (Ren et al., 2021)	3.95	—	0
Streamed, 2 slots	**1.93**	**2.05×**	**4.234**

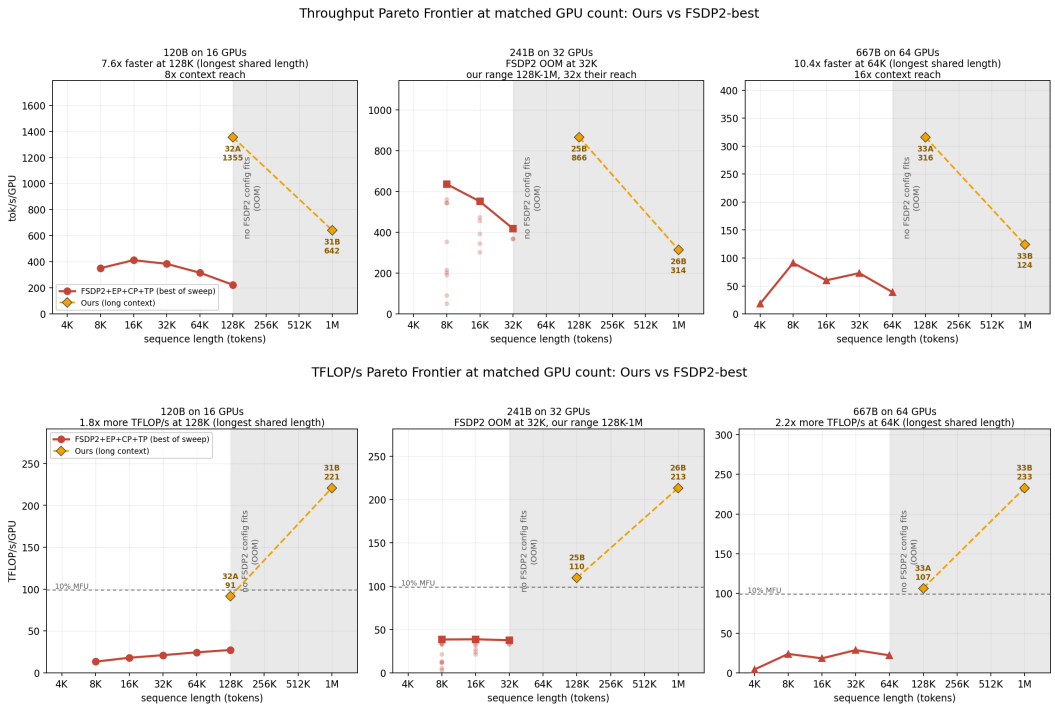


Figure 6: End-to-end training at matched model and GPU count. **Top:** per-GPU token throughput. **Bottom:** per-GPU TFLOP/s; the dashed line marks 10% model FLOP utilization. Solid lines show FSDP2-best, faint markers its dominated configurations, diamonds the full stack, and shading lengths where FSDP2 does not fit.

4 End-to-End Integration

We run the full stack under the hybrid module-specific regime of §3.5 on 120B, 241B, and 667B MoE models using 16, 32, and 64 H200 GPUs. FSDP2-best is the highest-throughput configuration in a sweep over FSDP2 with expert, context, and tensor parallelism (Zhao et al., 2023), CPU offloading, and matched by model and GPU count. Model dimensions are in §A.

Context reach and throughput. As shown in Figure 6 (top), FSDP2-best exhausts device memory past 128K tokens at the three scales, while the full stack trains at 1M tokens at all three, which is 8–32× the reach. It is also the faster of the two at the longest length FSDP2-best reaches, by 7.6× at 128K on 120B and 10.4× at 64K on 667B. On the TFLOPS/s axis (Figure 6 - bottom), the trend reverses because attention work per token grows with context. The full stack's per-GPU rate roughly doubles between 128K and 1M tokens at all three scales, from $\sim$ 100 to $\sim$ 233 TFLOP/s, while FSDP2-best stays below 40 anywhere in its sweep. At those same lengths, the full stack delivers 2.2× the baseline's rate at 667B scale.

Batch size. Device memory that a step does not commit to a fixed working set is available for tokens, and Figure 7 gives the two axes it can be spent on. The largest global batch that runs is 1.5M, 1.8M, and 3M distinct tokens per forward and backward pass, which is 12×, 7×, and 3× the

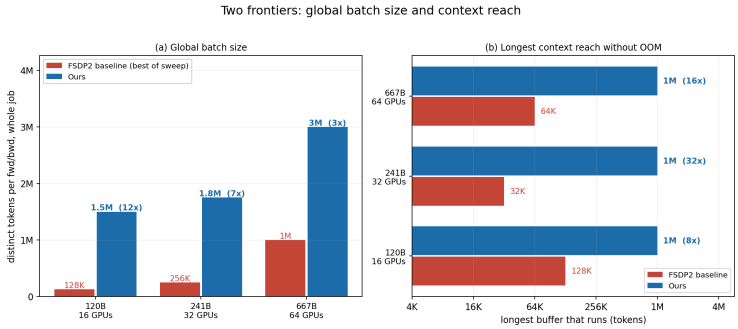


Figure 7: Batch and context frontiers at matched model and GPU count: **(a)** largest global batch and **(b)** longest context that fits. Multiples are relative to FSDP2-best.

baseline's, and the longest context that runs is 1M tokens at every scale. Every term of Equation (6) is fixed by model or launch configuration, so a workload that grows becomes more streamed work rather than a larger footprint. Training quality remains unchanged, as shown in §A.

5 Related Work

Parallelism and expert dispatch. Megatron-style tensor, sequence, and pipeline parallelism factor devices into a global grid (Shoeybi et al., 2019; Korthikanti et al., 2023), ZeRO shard model state (Rajbhandari et al., 2020; Zhao et al., 2023), expert parallelism partitions sparse experts (Rajbhandari et al., 2022), and Ulysses shards long sequences (Jacobs et al., 2023). Our hybrid module-specific parallelism regime assigns these schemes per module over overlapping groups of one rank set. FasterMoE, Lina, PipeMoE, and ScheMoE partition dispatch to overlap communication, choosing the partition count for bandwidth utilization (He et al., 2022; Li et al., 2023; Shi et al., 2023; 2024), which sizes the receive buffer before the partition is applied and leaves the footprint following the realized load. PipelinedLLEP instead caps each source's contribution to a chunk, which bounds receiver memory through checkpointed backward. Placement methods redistribute skew across devices (Zhai et al., 2023; Nie et al., 2023; Nguyen et al., 2026) and compose with the budget, since a placement fitted to observed co-selection statistics carries no guarantee for the next routing matrix.

Streamed losses, activations, and optimizer state. Cut Cross-Entropy evaluates the log-sum-exp on the fly in on-chip memory (Wijmans et al., 2025) and Liger-Kernel fuses the projection with cross-entropy over chunked inputs (Hsu et al., 2024), both resting on the online-softmax recurrence (Milakov and Gimelshein, 2018) behind FlashAttention (Dao et al., 2022), while Megatron's vocabulary-parallel cross-entropy shards the weight and reduces per-shard statistics across tensor-parallel ranks (Shoeybi et al., 2019). Each assumes away the case Ring-DTP addresses, since the fused kernels loop over a batch resident on one device and the vocabulary-parallel reduction needs a replicated batch, so no schedule is required; Ring Attention circulates key-value blocks around a ring (Liu et al., 2024), where no payload choice arises because only one operand can move. For activations, checkpointing and rematerialization reduce retained interiors (Chen et al., 2016; Jain et al., 2020; Kirisame et al., 2021), hooks and swapping relocate them (PyTorch Contributors, 2026a; Peng et al., 2020), and DeepSpeed (DeepSpeed Team, 2026) and PyTorch (PyTorch Contributors, 2026b) offload checkpoint regions wholesale, whereas SCO offloads a host-budget subset of checkpoint boundaries and prefetches them one layer ahead. ZeRO-Offload moves optimizer state and its update to lower tiers (Ren et al., 2021; Rajbhandari et al., 2021), while 8-bit quantization (Dettmers et al., 2022) and block-shared learning rates (Zhang et al., 2025) shrink it by changing numerics, which this setting excludes.

6 Conclusion

Together, the four operators bound memory for expert dispatch, vocabulary projection, checkpoint boundaries, and offloaded optimizer updates using only the launch configuration. Across 120B–667B MoE models, the stack trains at one-million-token context, 8–32$\times$ the reach of tuned FSDP2, with up to 10.4$\times$ its throughput and 12$\times$ its largest batch.

THE USE OF LARGE LANGUAGE MODELS

Large language models were used as a writing and coding aid. For the manuscript, they assisted with drafting and editing prose, tightening captions, and formatting references; every claim, number, and citation was checked against the source measurements and the cited papers by the authors. For the implementation, they assisted with routine code editing and test scaffolding; all operators, correctness tests, and measurements were designed, reviewed, and executed by the authors. Large language models were not used to generate research ideas, experimental designs, or results, and the authors take full responsibility for the content of this paper.

REFERENCES

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. URL https://arxiv.org/abs/1604.06174.

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

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. URL https://arxiv.org/abs/2412.19437.

DeepSpeed Team. Activation checkpointing: `cpu_checkpointing` and `contiguous_memory_optimization`. https://deepspeed.readthedocs.io/en/latest/activation-checkpointing.html, 2026. DeepSpeed documentation.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://arxiv.org/abs/2110.02861.

Wei Du, Shubham Toshniwal, Branislav Kisacanin, Sadegh Mahdavi, Ivan Moshkov, George Armstrong, Stephen Ge, Edgar Minasyan, Feng Chen, and Igor Gitman. Nemotron-Math: Efficient long-context distillation of mathematical reasoning from multi-mode supervision. *arXiv preprint arXiv:2512.15489*, 2025. URL https://arxiv.org/abs/2512.15489.

Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. MegaBlocks: Efficient sparse training with mixture-of-experts. In *Conference on Machine Learning and Systems (MLSys)*, 2023. URL https://arxiv.org/abs/2211.15841.

Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. FasterMoE: Modeling and optimizing training of large-scale dynamic pre-trained models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022. URL https://doi.org/10.1145/3503221.3508418.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient Triton kernels for LLM training. *arXiv preprint arXiv:2410.10989*, 2024. URL https://arxiv.org/abs/2410.10989.

Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale. In *Conference on Machine Learning and Systems (MLSys)*, 2023. URL https://arxiv.org/abs/2206.03382.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. DeepSpeed Ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023. URL https://arxiv.org/abs/2309.14509.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Conference on Machine Learning and Systems (MLSys)*, 2020. URL https://arxiv.org/abs/1910.02653.

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://arxiv.org/abs/2006.09616.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *Conference on Machine Learning and Systems (MLSys)*, 2023. URL https://arxiv.org/abs/2205.05198.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020. URL https://arxiv.org/abs/2006.16668.

Jiamin Li, Yimin Jiang, Yibo Zhu, Chuan Wang, and Hong Xu. Accelerating distributed MoE training and inference with Lina. In *USENIX Annual Technical Conference (ATC)*, 2023. URL https://www.usenix.org/conference/atc23/presentation/li-jiamin.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.06511.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.01889.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://arxiv.org/abs/1711.05101.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018. URL https://arxiv.org/abs/1805.02867.

Xuan-Phi Nguyen, Shrey Pandit, Austin Xu, Caiming Xiong, and Shafiq Joty. Least-loaded expert parallelism: Load balancing an imbalanced mixture-of-experts. *arXiv preprint arXiv:2601.17111*, 2026. URL https://arxiv.org/abs/2601.17111.

Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. FlexMoE: Scaling large-scale sparse pre-trained model training via dynamic device placement. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2023. URL https://doi.org/10.1145/3588964.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025. URL https://arxiv.org/abs/2508.10925.

Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020. URL https://doi.org/10.1145/3373376.3378505.

PyTorch Contributors. Automatic differentiation: Saved tensor hooks and `save_on_cpu`. https://docs.pytorch.org/docs/stable/autograd.html, 2026a. PyTorch documentation.

PyTorch Contributors. Activation checkpointing: `checkpoint_wrapper` and `offload_to_cpu`. https://docs.pytorch.org/docs/stable/distributed.checkpoint.html, 2026b. PyTorch distributed documentation.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020. URL https://arxiv.org/abs/1910.02054.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021. URL https://arxiv.org/abs/2104.07857.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *International Conference on Machine Learning (ICML)*, 2022. URL https://arxiv.org/abs/2201.05596.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *USENIX Annual Technical Conference (ATC)*, 2021. URL https://arxiv.org/abs/2101.06840.

Shaohuai Shi, Xinglin Pan, Xiaowen Chu, and Bo Li. PipeMoE: Accelerating mixture-of-experts through adaptive pipelining. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2023. URL https://doi.org/10.1109/INFOCOM53939.2023.10228874.

Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. ScheMoE: An extensible mixture-of-experts distributed training system with tasks scheduling. In *European Conference on Computer Systems (EuroSys)*, 2024. URL https://doi.org/10.1145/3627703.3650083.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. URL https://arxiv.org/abs/1909.08053.

Erik Wijmans, Brody Huval, Alexander Hertzberg, Vladlen Koltun, and Philipp Krähenbühl. Cut your losses in large-vocabulary language models. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2411.09009.

Mingshu Zhai, Jiaao He, Zixuan Ma, Zan Zong, Runqing Zhang, and Jidong Zhai. SmartMoE: Efficiently training sparsely-activated models through combining offline and online parallelization. In *USENIX Annual Technical Conference (ATC)*, 2023. URL https://www.usenix.org/conference/atc23/presentation/zhai.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik P. Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2406.16793.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023. URL https://arxiv.org/abs/2304.11277.

A EXPERIMENTAL SETUP

Each operator is measured on the live set it governs, so each benchmark is configured for that operator rather than for a shared end-to-end workload; Table 5 collects the settings. Every controlled benchmark runs on one node with 8 NVIDIA H200 GPUs and compares systems on byte-identical input in the same execution, so a comparison never spans two allocations of the machine. Latencies for the dispatch benchmark are means over ten measured forward calls after three warmup calls. Memory is the maximum PyTorch peak allocation across ranks for the dispatch and projection benchmarks, and the maximum 100 ms NVML sample across ranks after warmup for SCO, since no single counter captures allocator peaks, external runtime allocations, and host residency at once. Tensor totals count each allocation once.

Table 5: Configuration of each controlled component benchmark.

Benchmark	Configuration
Hardware	One node, eight NVIDIA H200 GPUs, intra-node NVLink, 2 TB host memory
Expert dispatch (PipelinedLLEP)	Two shapes: 32,768 tokens per rank at $H = I = 4096$ with top-4 and $c = 10{,}923$, and 65,536 tokens per rank at $H = 7168$, $I = 2048$ with top-8 and $c = 6554$; 128 experts, one eight-rank expert-parallel group, BF16; strided chunk membership; expert-matmul checkpointing; 384 MiB local-expert workspace
Vocabulary projection (Ring-DTP)	A distinct local batch of $N = 16{,}384$ per rank, $H = 7168$, $V = 200{,}000$, FP32; projection group $P \in \{4, 8\}$; auto-mode operand selection
Checkpoint offload (SCO)	gpt-oss-20b in BF16; sequence and expert parallelism of degree eight; 24 decoder layers exposing 47 checkpoint boundaries; 556,432 tokens per step at a configured ceiling of 557,056; pinned host budget swept over $\{0, 8, 16\}$ GiB and unlimited; depth-one restore prefetch
Offloaded optimizer (OffloadStreamAdamW)	gpt-oss-20b; fp32 master weights and moments in host memory on both sides; 100M-parameter buckets and $s = 2$ staging slots, selected from Table 14

Scaling workload. The end-to-end comparison of §4 is a pre-training workload, with all weights initialized from scratch, at 120B, 241B, and 667B parameters on 2, 4, and 8 nodes of the same type, that is 16, 32, and 64 GPUs. The three models share one architecture and differ only in depth: hidden width $H = 7168$, dense FFN intermediate width $18{,}432$, 384 routed experts of intermediate width $I = 2048$ with top-8 routing, and $V = 200{,}000$. The 120B, 241B, and 667B models have 7, 14, and 39 decoder layers. Routing is dropless, so every token reaches all eight of its experts, and load is balanced by the auxiliary-loss-free rule of DeepSeek-V3, which adds a per-expert bias to the routing scores and adjusts each bias at the end of every step according to that expert's load (DeepSeek-AI, 2024). In the layout of §A, expert shards span every rank, so $E_p = W$ is 16, 32, and 64 at the three scales, the projection group is $P = 8$, and D is set between 2 and 16 according to the context length. Both stacks run the same model at the same GPU count at each scale. FSDP2-best is selected per scale by sweeping FSDP2, tensor, context, and expert parallelism together with optimizer CPU offload on a TorchTitan-based stack (Zhao et al., 2023; Liang et al., 2025), and taking the highest-throughput configuration that fits; the whole sweep appears in Figure 6 as faint markers, and its per-length upper envelope is the solid line. The per-source token budget c is swept from 4096 to 32768 tokens per rank.

Rank layout and execution. The same W ranks are reused across three overlapping groupings, and every rank belongs to one group of each type. Each D-rank group exchanges sequence shards for attention-head shards at an attention block and restores the sequence layout afterward (Jacobs et al., 2023). Each E_p-rank group holds disjoint expert shards, each P-rank group holds disjoint vocabulary intervals, and ZeRO-3 shards dense weights across all W ranks (Rajbhandari et al., 2020). During forward, PipelinedLLEP dispatches each MoE layer in token chunks, SCO copies selected checkpoint boundaries to pinned host memory, and Ring-DTP visits every local batch with every vocabulary interval at the output head. Backward replays Ring-DTP, restores each offloaded boundary one

layer before recomputation, and recomputes each chunked MoE layer under its nested checkpoint. After backward and global-norm computation, OffloadStreamAdamW streams host-resident AdamW buckets through the GPU for update and write-back. The operators act on disjoint live sets and can be enabled independently.

Training quality. Each operator preserves forward values and gradients, so the composed stack trains to the same quality as a standard implementation given the same recipe. We check this end to end by supervised fine-tuning gpt-oss-20b at low reasoning effort on Nemotron-Math (Du et al., 2025), whose assistant responses are generated by gpt-oss-120b, at a context length of 128K, and evaluating Avg@8 on AIME 2025. Trained on the composed stack the model reaches 59.8%; trained on the FSDP2 baseline under the same data, schedule, and hyperparameters it reaches 59.6%; and gpt-oss-20b before fine-tuning reaches 45.2%.

B PIPELINEDLLEP

B.1 BACKWARD AND THE NESTED CHECKPOINT CONTRACT

The peak of a checkpointed step falls inside backward, so Equation (1) is only useful if it survives recomputation. Three properties deliver that.

The partition is recomputable, not saved. Chunk i holds token positions $\{i, i+K, i+2K, \dots\} \cap [0, N)$, a function of N and K alone, and K is reduced with a maximum over the expert-parallel group before any chunk collective is issued. A forward and its recomputation therefore cut the token axis identically without storing anything route-scale, and every rank runs the same number of chunks even when their token counts differ.

The inner checkpoint is reentrant so the outer one can see it. The expert matmul holds the $\Theta(kc_{\text{eff}}I)$ expert intermediate and the resident and imported expert weights the chunk needs, and it is wrapped in a *reentrant* checkpoint. Reentrant checkpointing routes every tensor argument through autograd's saved-tensor mechanism (PyTorch Contributors, 2026a), so when the decoder layer's own non-reentrant checkpoint is active its pack hook sees each of those tensors and replaces it with a placeholder. Those weights are then resident for one layer's forward rather than from that forward until the global backward. A non-reentrant inner checkpoint would instead keep the forward function and its arguments in an unpack closure holding strong Python references for the whole forward-to-backward lifetime; those references sit outside any saved-tensor mechanism, so no outer hook can intercept them, and with K chunks over L layers the expert weights would stay live for $\mathcal{O}(KL)$ cross-layer duration. Only the expert matmul is checkpointed, not the whole chunk, because placing all three stages of chunk i in one checkpoint call serializes them from Python's point of view and prevents the dispatch prefetch of chunk $i{+}1$; the cost is that dispatch outputs stay live across chunks, which Equation (1) already bounds.

Gradients accumulate once per layer. Dispatch and combine use all-to-all primitives whose backward is the symmetric inverse all-to-all, so gradients flow through the pipeline as ordinary autograd tensors and no chunk graph is retained past its own backward. Every chunk's expert-matmul backward accumulates into the same resident and imported expert-weight tensors, so the weight-migration backward fires exactly once per layer and expert-parameter gradients are returned once, preserving a single sharded-optimizer hook boundary. A scalar anchor returned by the weight-transfer function is added to the final output once rather than per chunk, which gives a rank that only sends foreign expert weights and never receives routes a path from the loss to its transfer.

Algorithm 1 PipelinedLLEP on one expert-parallel rank

Input: hidden states X, route weights A, expert ids E, local expert parameters Θ_{loc}, token budget c, ceiling $K_{\max}$
$K \leftarrow \min(\text{AllReduceMax}(\lceil N/c \rceil), K_{\max})$
run the LLEP load measurement and assignment plan once; migrate expert weights if the plan requires it
$\pi \leftarrow$ per-route destinations from the layer-wide plan; $S \leftarrow \text{AllGather}$ of the (K, E_p) send splits
$Y \leftarrow 0$; issue dispatch of chunk 0
for $i = 0, \ldots, K-1$ **do**
    $\mathcal{T}_i \leftarrow \{i, i+K, i+2K, \ldots\} \cap [0, N)$ ▷ strided membership
    $(X_i, A_i, E_i) \leftarrow$ gather rows $\mathcal{T}_i$ in destination order under π
    $(\widehat{X}_i, \widehat{A}_i, \widehat{E}_i) \leftarrow \text{A2A}((X_i, A_i, E_i); S_i, S_i^{\top})$ ▷ dispatch, one coalesced group
    $\widehat{Y}_i \leftarrow \text{ckpt}_{\text{reentrant}}\left[\text{LocalExperts}(\widehat{X}_i, \widehat{A}_i, \widehat{E}_i; \Theta_{\text{loc}})\right]$
    issue dispatch of chunk $i{+}1$ on the dispatch stream
    $Y_i \leftarrow \text{A2A}(\widehat{Y}_i; S_i^{\top}, S_i)$ ▷ combine
    $Y[\mathcal{T}_i] \leftarrow$ top-k reduction of Y_i; release chunk-local tensors
Output: Y, and the anchor that drives the weight-transfer backward once

PyTorch-style reproduction sketch. Alg. 2 fixes the execution choices used in the paper and shows the tensor operations more explicitly. It assumes three short primitives with the following contracts. `llep_lla_plan()` returns a destination rank for every route and the associated expert-weight moves, using the published LLEP LLA plan (Nguyen et al., 2026); we do not repeat that planner here. `move_experts_async()` batches those moves on a communicator distinct from the expert-parallel group and returns the imported tensors, one completion handle, and a zero-valued autograd anchor. `coalesced_a2a()` is a variable-split, autograd-aware all-to-all whose backward swaps the send and receive splits; its three native-dtype payloads are launched as one communication group without concatenation. The weight tuple contains only tensor arguments (gate/up weight and bias, down weight and bias, and expert-id lookup tensors), so the reentrant checkpoint does not capture weights in a Python closure.

For clarity, `expert_ffn` below gives exact reference math as an ordinary per-expert loop. To reproduce the reported kernel path, partition its rows into resident and imported experts and replace each projection loop with two `torch._grouped_mm` calls, concatenating their outputs. This preserves the shown math while avoiding a combined resident–imported weight stack; add the corresponding bias in-place after each grouped matrix multiplication.

Algorithm 2 PyTorch-like pseudocode for the paper configuration of PipelinedLLEP

```
def expert_ffn(z, expert, score, *weight_tensors):
    W = views_by_global_expert_id(*weight_tensors); limit, alpha = 18.0, 0.5
    y = torch.empty_like(z)
    for e in expert.unique().tolist():
        m = expert == e
        gate_up = z[m] @ W[e].gate_up + W[e].b_gate_up
        gate, up = gate_up[..., ::2], gate_up[..., 1::2]
        gate = gate.clamp(max=limit); up = up.clamp(min=-limit, max=limit)
        h = (up + 1) * gate * torch.sigmoid(alpha * gate)
        y[m] = h @ W[e].down + W[e].b_down
    return y * score[:, None]
def dispatch(rows, x, expert, score, dst, send_n, recv_n, G, stream):
    with torch.cuda.stream(stream):
        order = dst[rows].flatten().argsort()       # group routes by receiver
        source_row = order // expert.shape[1]
        send = (x[rows][source_row],
                score[rows].flatten()[order],
                expert[rows].flatten()[order])
        recv = coalesced_a2a(send, send_n, recv_n, G)
        ready = torch.cuda.Event(); ready.record(stream)
    return (*recv, order.argsort(), ready)
def pipelined_llep(x, expert, router_score, W, G, weight_group):
    N, k, K = x.shape[0], expert.shape[1], 10
    rank, P = dist.get_rank(G), dist.get_world_size(G)
    score = router_score.gather(1, expert.long()).to(x.dtype)
    count = torch.bincount(expert.flatten(), minlength=router_score.shape[1])
    dist.all_reduce(count, group=G)
    dst, moves = llep_lla_plan(expert, count, G, imbalance_threshold=1.1)
    rows = [torch.arange(i, N, K, device=x.device) for i in range(K)]
    send_n = torch.stack([torch.bincount(dst[r].flatten(), minlength=P)
                          for r in rows])
    gathered = [torch.empty_like(send_n) for _ in range(P)]
    dist.all_gather(gathered, send_n, group=G)
    recv_n = torch.stack(gathered)[:, :, rank].T.contiguous()
    send_n, recv_n = send_n.tolist(), recv_n.tolist()
    compute = torch.cuda.current_stream(); stream = torch.cuda.Stream()
    stream.wait_stream(compute)
    current = dispatch(rows[0], x, expert, score, dst,
                       send_n[0], recv_n[0], G, stream)
    dist.barrier(group=weight_group)                        # A(0) keeps running
    imported, weight_work, anchor = move_experts_async(
        W, moves, group=weight_group)                        # separate NCCL
    parts = []
    for i in range(K):
        following = (dispatch(rows[i + 1], x, expert, score, dst,
                     send_n[i + 1], recv_n[i + 1], G, stream)
                     if i + 1 < K else None)                 # depth one
        if i == 0:                                           # A(0), A(1) queued
            weight_work.wait()
            weights = explicit_tensor_tuple(W, imported)     # no stack
        rx, ra, re, undo, ready = current
        compute.wait_event(ready)
        for t in (rx, ra, re): t.record_stream(compute)
        routed = checkpoint(expert_ffn, rx, re, ra, *weights,
                            use_reentrant=True)
        returned = a2a_autograd(routed, recv_n[i], send_n[i], G)
        parts.append(returned[undo].reshape(len(rows[i]), k, -1).sum(1))
        current = following
    order = torch.cat(rows)
    y = torch.empty_like(x).index_copy(0, order, torch.cat(parts))
    return y + anchor                                        # once per layer
```

Which gradient the partition touches. For $g_t = \partial\mathcal{L}/\partial y_t$, route weight α_{tj}, and expert function f_e,

$$\frac{\partial\mathcal{L}}{\partial\alpha_{tj}} = g_t^\top f_{e(t,j)}(x_t), \qquad \frac{\partial\mathcal{L}}{\partial x_t} = \sum_j \alpha_{tj} J_{f_{e(t,j)}}(x_t)^\top g_t, \qquad \frac{\partial\mathcal{L}}{\partial\theta_e} = \sum_{(t,j):e(t,j)=e} \alpha_{tj} \frac{\partial f_e(x_t)}{\partial\theta_e}^\top g_t. \tag{7}$$

Only $\partial\mathcal{L}/\partial\theta_e$ has its summation order changed, since it is the one sum split across chunks and reassembled in the shared accumulator of Alg. 1. At $K = 1$ that sum has a single term and the expert-parameter gradients are bitwise identical to LLEP's. Hidden-state and router gradients are never split.

Cost accounting. Route payload is unchanged in both directions: each route crosses the interconnect once per pass, and the additions are per-chunk metadata exchanges that are $\mathcal{O}(1)$ per route rather than $\mathcal{O}(H)$. Two costs scale with K. Collective launches grow linearly, since dispatch issues one coalesced group of three native-dtype all-to-alls, combine issues one, and backward issues their symmetric inverses. The expert matmul is evaluated three times per update under decoder-level checkpointing, namely the original forward, the decoder recomputation, and the inner recomputation, against twice for a checkpointed unpartitioned layer; dispatch and combine are evaluated twice either way.

B.2 Routing Profiles

A profile is a pair r/h in which r is the hot-token percentage and h the number of hot experts. Experts $0, \ldots, h-1$ are hot and the remaining $E - h$ are cold. The first $\lfloor rN \rfloor$ token positions are hot and select experts $\{j \bmod h : j = 0, \ldots, k-1\}$, so a hot token's k routes cycle over the hot set and collapse onto $\min(k, h)$ distinct experts. Each remaining token i selects cold experts $\{h + ((i+j) \bmod (E-h)) : j = 0, \ldots, k-1\}$, which walks the cold set and spreads the residual load uniformly. The balanced profile assigns token i the experts $\{(i+j) \bmod E : j = 0, \ldots, k-1\}$. Routing weights are $1/k$ on every selected expert. Skew rises as r rises and as h falls, and $h \geq k$ against $h < k$ is the qualitative boundary, since below it the hot set cannot absorb k distinct routes and repeats. We sweep $r/h \in \{30/16, 50/16, 80/16, 95/16, 30/4, 50/4\}$ plus balanced.

B.3 Per-Profile Measurements

Unless a table states otherwise, every PipelinedLLEP entry uses strided chunk membership and expert-matmul checkpointing, and the chunk count is stated per table. Peak memory is deterministic in the configuration at both shapes: three repeats of the 65K sweep reproduce all seven peaks to the milli-GiB, and latency spread across those repeats is at most 2.6%.

A chunk count tuned for the top of the range cuts a 128-token micro-batch into ten pieces and pays ten full collective rounds for roughly one token each, which costs 0.45–$0.48\times$; the budget derives $K = 1$ instead and runs at 1.60–$2.96\times$, faster than LLEP itself, because the single chunk still uses the coalesced single-group all-to-all. Peak reduction tracks the size of the unpartitioned live set, rising from 9.0% at 128 tokens to 65.9% at 131,072, since below roughly 8,192 tokens per rank the dispatch buffers are small next to the resident expert weights.

Strided membership runs at 1.03–$1.35\times$ the speed of contiguous membership with 0.03–1.37 GiB lower peak, and the send-ratio columns give the mechanism. The same counter reports $\max_d R_d^{(i)}$ against the bound of Equation (1). Strided membership realizes 12.5% of it, which is the even share $1/E_p$, at every chunk size and on every profile, while contiguous membership reaches 17.1% on the $50\%/4$ profile. The bound is therefore respected everywhere and attained nowhere: all-to-all buffers are sized from realized split counts, so the peak follows kc_{eff} and the bound is the guarantee that no router can move it rather than a description of the traffic.

Table 9 confirms that the reduction is not an artifact of where a forward-only measurement stops: including backward, the reduction at the reference shape is unchanged at 56.9–59.3%. The 32K row shows the other side of the trade. At $K = 10$ that shape sits past the flat region of Figure 8b, so the chunk count that is free at 65K costs 0.87–$0.90\times$ there, which is what deriving $K = 8$ from the budget avoids at that length.

Table 6: Routing-skew sweep at both shapes. Entries are mean forward latency in ms / maximum peak allocated memory across ranks in GiB, from executions in which the methods compared run on identical input. Speedup and peak reduction are against LLEP. At 65K the sweep was repeated three times and each row reports the repeat with the lowest PipelinedLLEP latency together with that repeat's own LLEP measurement, so the columns are paired; §B.5 explains why the lowest repeat is representative. At the 65K shape standard expert parallelism runs out of memory on every skewed profile; on the balanced profile LLEP performs no relocation, so the two coincide.

Routing	Standard EP (ms / GiB)	LLEP (ms / GiB)	PipelinedLLEP (ms / GiB)	Speed vs. LLEP	Peak saved
32,768 tokens per rank, $H = I = 4096$, top-4, $c = 10{,}923$ ($K = 3$)					
Balanced	40.82 / 21.347	47.95 / 24.101	44.74 / 17.606	**1.07×**	**26.9%**
30% / 16	89.39 / 32.549	53.14 / 24.476	51.44 / 18.460	1.03×	24.6%
50% / 16	143.12 / 45.351	53.74 / 24.476	54.57 / 18.460	0.98×	24.6%
80% / 16	225.06 / 64.556	53.89 / 24.664	53.90 / 18.553	1.00×	24.8%
95% / 16	267.08 / 74.154	54.34 / 24.664	53.93 / 18.553	1.01×	24.8%
30% / 4	107.28 / 36.885	53.49 / 24.664	52.27 / 18.554	1.02×	24.8%
50% / 4	157.39 / 48.445	53.61 / 24.664	53.97 / 18.554	0.99×	24.8%
65,536 tokens per rank, $H = 7168$, $I = 2048$, top-8, $c = 6554$ ($K = 10$)					
Balanced	177.82 / 52.660	177.82 / 52.660	161.00 / 21.430	**1.10×**	**59.3%**
30% / 16	OOM	182.97 / 52.906	176.62 / 22.772	1.04×	57.0%
50% / 16	OOM	186.07 / 52.906	182.64 / 22.773	1.02×	57.0%
80% / 16	OOM	189.14 / 52.988	180.30 / 22.861	1.05×	56.9%
95% / 16	OOM	182.13 / 52.988	178.57 / 22.855	1.02×	56.9%
30% / 4	OOM	185.78 / 52.906	180.59 / 22.774	1.03×	57.0%
50% / 4	OOM	186.79 / 52.906	184.69 / 22.778	1.01×	57.0%

Table 7: Micro-batch length ladder at the 65K shape under one token budget, $c = 4096$ with $K_{\max} = 10$, against a chunk count fixed at $K = 10$. Speedup and peak reduction compare PipelinedLLEP with LLEP at the same length, with LLEP re-measured in the same execution; ranges are over the balanced and $80\%/16$ profiles. Where a length was executed more than once we report the lowest speedup observed, so the ranges are the pessimistic reading. The ceiling binds above $N = cK_{\max} = 40{,}960$, where the two configurations coincide.

	Token budget $c = 4096$			Fixed $K = 10$	
Tokens N	K	Speedup	Peak saved	Speedup	Peak saved
128	1	**1.60–2.96×**	9.0–11.0%	0.45–0.48×	9.4–11.4%
1,024	1	**1.35–2.17×**	10.4–11.1%	0.47–0.56×	14.0–15.6%
8,192	2	0.98–1.09×	14.3–17.0%	0.80–0.82×	28.6%
32,768	8	1.02–1.09×	44.9–47.6%	0.98–1.06×	47.2–49.4%
65,536	10	1.04–1.10×	56.9–59.3%	*coincides*	
131,072	10	1.07–1.10×	63.4–65.9%	*coincides*	

B.4 Choosing the Token Budget

B.5 Two Latency Modes at the 65K Shape

Forward latency at the 65K shape is bimodal, and the modes are far enough apart to matter for how the timing tables should be read. Five back-to-back passes over the seven profiles at $K = 16$ separate them cleanly: in the fast mode every profile reproduces to within 0.5% across passes, which is tighter than any other latency we measure, while the slow mode adds 12–56% and never subtracts. The mode is not a property of the routing profile, since the same profile appears in either mode across passes, and it is strongly a property of the chunk count. Over everything measured at this shape, $K = 10$ is slow in 2 of 56 profile measurements against 21 of 84 for $K = 16$.

Table 8: Chunk membership at 65,536 tokens per rank, $K = 10$, all other settings identical. *Send ratio* is the largest destination send divided by the mean send within a chunk, taken as the maximum over chunks and measured by a per-chunk counter on the dispatch all-to-all. Membership changes neither the token set nor any route's destination, so the two policies differ only in schedule quality.

	Latency (ms)		Peak (GiB)		Send ratio	
Routing	Strided	Contig.	Strided	Contig.	Strided	Contig.
Balanced	**161.2**	218.2	**21.430**	21.464	1.00	1.02
95% / 16	**178.6**	184.9	**22.855**	23.575	1.35	2.63
80% / 16	**180.6**	209.7	**22.861**	23.514	2.40	4.00
50% / 4	**190.8**	217.0	**22.778**	24.151	2.29	6.00

Table 9: Forward and backward together, c chosen to give $K = 10$ at each shape, routing profiles balanced / $80\%/16$. Peak is the maximum allocated memory across ranks over the combined pass.

Shape	LLEP (GiB)	PipelinedLLEP (GiB)	Peak saved	Speedup
65K tok, H=7168, top-8	52.660 / 52.988	**21.430 / 22.861**	**59.3 / 56.9%**	1.11 / 1.04×
32K tok, H=4096, top-4	23.850 / 24.413	**14.902 / 15.382**	**37.5 / 37.0%**	0.87 / 0.90×

The mechanism is the host-side launch pattern. Enabling a per-chunk diagnostic counter, which inserts one host-side synchronization per chunk and changes nothing about the computation, moves the slow mode off two profiles and onto a third, reproducibly in both passes of both conditions. For a given chunk count, routing profile, and host synchronization structure the operator enters one mode and stays in it. Two consequences follow. The representative latency for a configuration is the fast mode, so Table 6 reports the lowest of three repeats at 65K. And $K_{\max} = 10$ keeps the schedule in the region where the slow mode is rare, which is a second reason for the ceiling alongside the flat-region argument of Figure 8b. The single-execution points of Figure 8b at $K \in \{2, 3, 5\}$ are exposed to the effect: the hump at $K = 3$ is $1.75\times$ the $K = 10$ point, which exceeds the slow mode's range, so the ordering of those points against $K = 10$ holds while their magnitudes are not tight.

C RING-DTP

Each rank holds a distinct local batch, and the meetings visit every batch with every vocabulary shard exactly once. Target-column ownership is exclusive: exactly one shard interval contains a given target index, so exactly one meeting contributes y_t and no reduction over duplicated contributions is needed. Backward replays the same meetings from the saved per-token state, forms the softmax residual for one strip at a time, and contracts it into input and weight gradients, so neither pass stores a full logit or logit-gradient tensor.

The benchmark gives each rank a distinct local batch with $N = 16{,}384$, uses $H = 7{,}168$ and $V = 200{,}000$ in FP32, and measures hidden states to target-token log-probabilities followed by backward. The standard reference stores the full projection weight and materializes full logits and log-softmax.

D SCO

SCO walks the checkpointed layers in forward order and offloads each boundary that still fits in the host budget. Selected boundaries are copied asynchronously to capped pinned memory, and the device copy is freed. Backward visits layers in reverse. While layer ℓ recomputes from h_ℓ, the next boundary $h_{\ell-1}$ is copied back on a separate stream, so at most two restores occupy the device. Stream events preserve producer and consumer order. An allocation that would exceed the cap is left on the device.

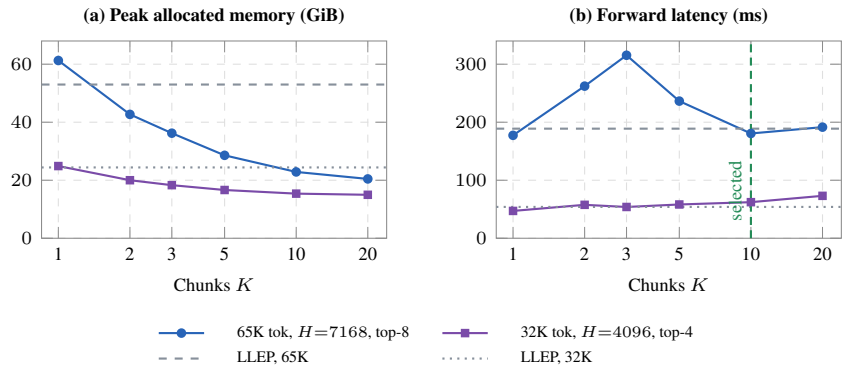


Figure 8: Chunk count against peak memory and forward latency at both shapes, routing profile $80\%/16$, with LLEP at the same shape as the horizontal reference. Peak falls monotonically in K with decaying marginal return: at 65K the first ten chunks remove $38.4\,\mathrm{GiB}$ and the next ten remove $2.4\,\mathrm{GiB}$. Latency does not follow the same shape, so K is selected from this curve rather than minimized. At 65K the points $K \in \{2, 3, 5\}$ are slower and larger than $K = 10$, while $K \in [10, 20]$ is flat to within 6%; at 32K, latency rises steadily past $K = 3$. $K = 1$ separates the schedule from the chunking: with one chunk the operator holds $8.3\,\mathrm{GiB}$ more than LLEP at 65K, because a single chunk adds a copy of the route-shaped tensors and partitions nothing. Each point is one execution of ten timed forward calls; LLEP is re-measured inside every execution and its reference line is the mean of those six measurements, which span 0.3%.

Algorithm 3 Ring-DTP forward on rank r of a projection group of size P

Input: local batch X_r of N tokens, targets t_r, weight shard W_r over the vocabulary interval $\mathcal{V}_r$
function FOLD(state S, strip Y, targets t, interval $\mathcal{V}$)
  for each token n **do**
    $(m, z, y) \leftarrow S_n$
    $m' \leftarrow \max\big(m, \max_v Y_{n,v}\big), \qquad z' \leftarrow e^{m-m'} z + \sum_v e^{Y_{n,v}-m'}$
    if $t_n \in \mathcal{V}$ **then** $y \leftarrow Y_{n,t_n}$
    $S_n \leftarrow (m', z', y)$
  return S
$S \leftarrow \text{FOLD}((-\infty, 0, 0),\ X_r W_r,\ t_r,\ \mathcal{V}_r)$
if $N > V/P$ **then**
  $W \leftarrow W_r$
  for $i = 1, \ldots, P-1$ **do**
    $q \leftarrow (r - i) \bmod P$
    send W to rank $r{+}1$, receive W_q from rank $r{-}1$, and set $W \leftarrow W_q$
    $S \leftarrow \text{FOLD}(S,\ X_r W,\ t_r,\ \mathcal{V}_q)$, then release the strip
else
  $(X, t, \widetilde{S}) \leftarrow (X_r, t_r, S)$
  for $i = 1, \ldots, P-1$ **do**
    send $(X, t, \widetilde{S})$ to rank $r{+}1$ and receive the next $(X, t, \widetilde{S})$ from rank $r{-}1$
    $\widetilde{S} \leftarrow \text{FOLD}(\widetilde{S},\ XW_r,\ t,\ \mathcal{V}_r)$, then release the strip
  send $\widetilde{S}$ to rank $r{+}1$ and receive S from rank $r{-}1$
Output: the negative log-likelihood $m + \log z - y$ of every token, from its state $S_n = (m, z, y)$

The host cap is a correctness condition rather than a comfort margin, because a rank that raised while the others entered a collective would deadlock. Setting it at or above $|\mathcal{S}| N_{\max} H b$ makes the check pass on every rank independently of what the others hold.

The matched benchmark uses gpt-oss-20b in BF16 on one eight-H200 node with sequence and expert parallelism of degree eight. Its 24-layer decoder exposes 47 checkpoint boundaries across the hidden and residual streams. Every policy processes the same 556,432 tokens per measured step at a

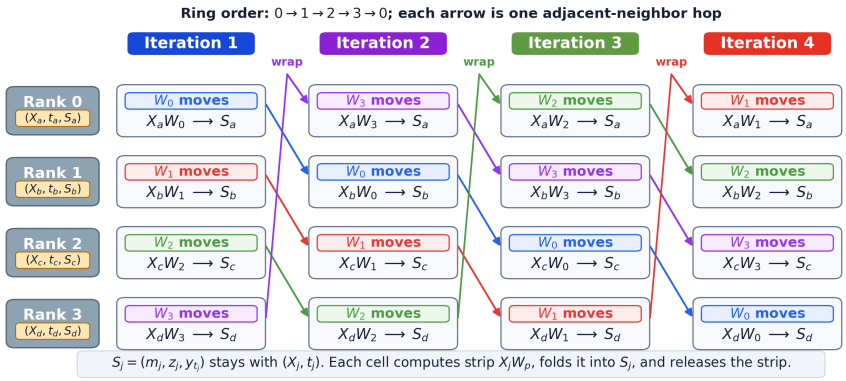


Figure 9: The move-weights schedule at $P = 4$. Rank r retains its batch (X_r, t_r) and running statistics S_r while each vocabulary block W_p advances one hop per iteration. A meeting computes X_rW_p, folds the strip into S_r, and releases it, so after P meetings every S_r is complete on its home rank and forward needs no return hop.

Table 10: Peak allocation for the vocabulary projection as the Ring-DTP group grows. Projection peaks include target-token log-probability computation and backward; the increment subtracts the measured resident weight-plus-input footprint. Changes are relative to the standard dense projection. Matched forward latency at the same shape is 989.8 against 1035.7 ms at $P = 4$ and 1001.2 against 1052.2 ms at $P = 8$.

P	Metric	Standard	Ring-DTP	Change
4	Projection weight	5.3406 GiB	1.3351 GiB	75.0% lower
	Resident weight + input	5.8409 GiB	1.8353 GiB	68.6% lower
	Projection F+B peak	42.4620 GiB	11.5194 GiB	72.9% lower
	Increment over resident	36.6212 GiB	9.6841 GiB	73.6% lower
8	Projection weight	5.3406 GiB	0.6676 GiB	87.5% lower
	Resident weight + input	5.8409 GiB	1.1678 GiB	80.0% lower
	Projection F+B peak	42.4620 GiB	**7.3222** GiB	**82.8% lower**
	Increment over resident	36.6212 GiB	6.1545 GiB	83.2% lower

configured ceiling of 557,056 and runs ten complete steps, the first two excluded as warmup, with the same stateless optimizer on both sides.

A session's boundaries all come from one micro-batch, so its pinned total inherits that micro-batch's length. Each session releases its pinned storage on completion, which is what makes the cap per-session rather than cumulative; otherwise the sessions comprising one update would accumulate.

Table 11: Matched forward latency at the shape of Table 10. Local batch is per GPU.

Local batch	P	Standard (ms)	Ring-DTP (ms)	Slowdown
16,384	4	989.8	1035.7	4.6%
16,384	8	1001.2	1052.2	5.1%

Algorithm 4 SCO over the checkpointed layers of one session

Input: checkpoint boundaries $h_1, \ldots, h_L$ in forward order, pinned host budget B
$\mathcal{S} \leftarrow \emptyset$
for $\ell = 1, \ldots, L$ **do**
    if h_ℓ fits in the unused part of B **then**
        copy h_ℓ to pinned host memory on the copy stream, then free the device copy
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{\ell\}$
if $L \in \mathcal{S}$ **then** start the restore of h_L on the copy stream
for $\ell = L, \ldots, 1$ **do**
    if $\ell \in \mathcal{S}$ **then** wait for the restore of h_ℓ
    if $\ell{-}1 \in \mathcal{S}$ **then** start the restore of $h_{\ell-1}$ on the copy stream
    recompute layer ℓ from h_ℓ and run its backward
    if $\ell \in \mathcal{S}$ **then** free the restored h_ℓ
release the pinned storage of $\mathcal{S}$

Table 12: Matched SCO budget sweep at 557,056 configured tokens. Logical payload is selected boundary storage and the pinned column includes host allocator size class rounding; both are per rank. Peak node RAM includes the common process baseline. Memory columns are GiB, step time is seconds, throughput is tokens/s/GPU.

Policy	Bound.	Logical	Pinned	Peak HBM	Node RAM	Step	Throughput
Off	0 / 47	0	0	139.790	402.517	26.3412	2,641
8 GiB	21 / 47	7.82–7.84	10.5	133.546	487.990	25.8415	**2,692**
16 GiB	42 / 47	15.63–15.69	21.0	125.677	573.473	25.8806	2,687
Full	47 / 47	17.49–17.55	23.5	**123.728**	593.410	25.8844	2,687

Table 13: Capacity on the separate 32,768-token search grid. "Largest clean" has no allocator-cache-flush warning; "largest completion" finishes all ten steps with warnings. The adjacent first failure brackets the marginal boundary, and gain is computed between marginal completions.

Policy	Largest clean	Largest completion	First failure	Marginal gain	Failed alloc.
Off	458,752	557,056	589,824	–	27.61 GiB
8 GiB	557,056	589,824	622,592	5.88%	29.03 GiB
16 GiB	557,056	622,592	655,360	11.76%	30.59 GiB
Full	**622,592**	**655,360**	688,128	**17.65%**	32.17 GiB

E OFFLOADSTREAMADAMW

A bucket co-groups a parameter shard, its fp32 master value and two moments, its final gradient, and the refreshed bf16 working weight. The fused kernel applies the same bias correction, weight decay, and update order as the reference implementation (Loshchilov and Hutter, 2019), and buckets are built once from parameter order, so ownership does not change between steps. Beginning a bucket during backward instead of after it additionally requires the bucket's gradient to be final and any cross-bucket transform to be known; global gradient clipping supplies the latter only once the global norm exists. The per-bucket and predictive clipping modes that lift this constraint change the update rule and sit outside the exactness claim.

Algorithm 5 OffloadStreamAdamW on one rank

1: **Input:** host-resident fp32 master weights and moments, final gradients, bucket size β, staging slots s
2: partition the rank's parameter shard into buckets $g = 1, \ldots, G$ of at most β parameters
3: **for** $g = 1, \ldots, G$ **do**
4: wait for a free staging slot and take it
5: on the transfer stream, copy bucket g's master weight, moments, and gradient into the slot
6: on the compute stream, apply the fused AdamW update and refresh the bf16 working weight
7: on the write-back stream, copy the updated fp32 master weight and moments to the host
8: return the slot to the free pool when its write-back event completes

E.1 DEEPSPEED RE-IMPLEMENTATION

The hybrid layout assigns an ownership group G_p to every parameter p. Shared parameters use all W ranks, an expert parameter uses the ranks that replicate that expert shard (possibly a singleton), and a tensor-parallel output shard uses the ranks that replicate that shard. Before `deepspeed.initialize`, create every group in the same order on every rank and assign the narrow groups first with `deepspeed.zero.Init`; a final all-rank `Init` claims only unassigned parameters. Parameter replacement and checkpoint restore must preserve this group metadata.

Our DeepSpeed v0.15.0 port changes the following paths for ZeRO-3 with `reduce_scatter=true`:

1. In `runtime/zero/partition_parameters.py`, split mixed parameter lists by process-group identity before flattening or gathering, and derive buffer size, partition count, rank, and collective group from that subgroup. A singleton gather is a no-op.
2. In `runtime/zero/partitioned_param_coordinator.py`, apply the same split to every coalesced forward/backward fetch and associate its handle only with parameters in that subgroup.
3. In `runtime/zero/stage3.py`, group persistent-parameter gathers and repartitions by G_p. Split backward gradient buckets the same way, sort each subgroup by stable ZeRO parameter id, and use the rank inside G_p when storing a reduce-scatter partition.
4. In `runtime/comm/coalesced_collectives.py`, add an `average` argument to reduce-scatter and its fallbacks; `average=false` performs an unscaled sum.
5. In `runtime/engine.py`, expose and validate the optimizer's reduction mode so an unpatched or averaging build fails before backward.

No flattened buffer or collective may mix ownership groups, and all members of one group must issue the same parameter order, shapes, dtypes, and padding.

For local loss numerator S_r and global denominator Z, Stage 3 must produce

$$g_p = \sum_{r \in G_p} \frac{\partial}{\partial p} \left(\frac{S_r}{Z} \right). \tag{8}$$

We therefore enable both paper modes before importing DeepSpeed:

```
DS_PERSISTENT_BY_PROC_GROUP=1  DS_GRAD_SUM_BY_PROC_GROUP=1.
```

Stage 3 reduce-scatters each homogeneous bucket with `average=false`, bypassing legacy predivide and postscale compensation. A singleton expert keeps its local gradient, expert replicas sum only within their replication group, and shared parameters sum over all ranks. The global loss denominator and gradient-accumulation weight are applied exactly once before backward. After reduce-scatter, each stored partition is unique, so the global norm is the all-rank sum of squared partition norms with no process-group-size correction.

E.2 Streamed CPU Adam in DeepSpeed

The paper path adds `runtime/zero/streaming_optim.py` and modifies `runtime/zero/stage3.py`. It is selected by `DS_OPTIMIZER_COMPUTE=gpu_stream` and is enabled only for ZeRO-3 with BF16 parameters, `DeepSpeedCPUAdam`, optimizer offload to CPU, GPU-resident parameter partitions, and no NVMe optimizer swap. We request a 10^8-parameter ZeRO subgroup and set `DS_OPTIMIZER_STREAM_SLOTS=2`; because a parameter is not split at a subgroup boundary, a slot is sized to the largest resulting subgroup rather than exactly 10^8 parameters.

Streaming engine. In `streaming_optim.py`, preallocate s GPU slots, each containing four fp32 buffers of the maximum subgroup length: master weight, first moment, second moment, and gradient. Each submission is ordered by CUDA events through three dedicated streams. The H2D stream copies those four arrays from pinned CPU memory; the compute stream calls the same multi-tensor FusedAdam kernel with the subgroup's learning rate, Adam coefficients, epsilon, weight decay, bias correction, and step count, then casts the updated master weight directly into the existing BF16 ZeRO partition; the D2H stream writes only the updated master weight and two moments back to their original CPU arrays. The gradient is not copied back because the next backward overwrites it. If all slots are busy, submission waits for the oldest D2H event and reuses that slot, providing bounded back-pressure and a fixed $16s\beta_{\max}$-byte HBM staging cost.

ZeRO-3 integration. In `stage3.py`, read the mode before building the fp32 partitions. Pin each CPU master and final-gradient buffer, lazily create the two CPUAdam moment arrays in pinned memory, and re-pin moments restored by checkpoint loading. After ZeRO has formed all flattened subgroups, construct the streaming engine once using their maximum length. The ordinary step prefix is unchanged: partition gradients, detect overflow, compute the global norm, and call `unscale_and_clip_grads` on every subgroup before submission. For a CPU subgroup, package references to its four CPU arrays, BF16 destination, hyperparameters, and pre-increment step; submit it instead of calling `CPUAdam.step()`, and increment CPUAdam's stored step counter. Since the engine already performs fp32-to-BF16 cast-back, the normal per-subgroup copy is skipped, although ZeRO still unflattens the updated partition metadata.

After the last subgroup, `flush()` waits for all D2H completions and makes the default stream wait for the compute stream. This single fence is required before CPU gradient buffers are reset, checkpoints read the host master and moments, persistent parameters are gathered, or the next forward reads the BF16 partitions. CPUAdam's host tensors and step counters remain the canonical checkpoint representation; only the location and scheduling of the AdamW arithmetic changes.

Gradient handoff. The engine receives the final ZeRO partition satisfying Equation (8), after loss unscaling and global clipping. OffloadStreamAdamW performs no further reduction, scaling, or clipping.

E.3 Bucket and Slot Sweep

All measurements use gpt-oss-20b on eight H200 GPUs, against the AVX CPU Adam kernel of ZeRO-Offload (Ren et al., 2021) at 3.95 s per optimizer step. The selected 100M-parameter, two-slot configuration averages 1.93 s (Table 4). Two axes are swept: the requested bucket size, which sets how many parameters cross the link at once, and the number of staging slots, which sets how deep the transfer queue runs.

Table 14: Bucket-size and slot-count sweep. Times are mean optimizer-step latency and staging is resident memory per GPU. The 100M bucket wins on both axes, and two slots stay within 1.1% of three at every bucket size while using a third less staging memory. A deeper queue does not help: step time is flat from two to five slots while each additional slot adds one bucket's worth of staging.

Requested bucket	Slots	Groups	Actual maximum	Time (s)	Staging (GiB)
100M	2	26	142.076M	**1.930**	**4.234**
100M	3	26	142.076M	1.951	6.351
150M	2	14	208.431M	2.204	6.212
150M	3	14	208.431M	2.225	9.318
200M	2	14	244.938M	2.240	7.300
200M	3	14	244.938M	2.252	10.950
250M	2	10	308.586M	2.282	9.197
250M	3	10	308.586M	2.282	13.795