CHUNKOPD: EFFICIENT STRICT ON-POLICY DISTILLATION VIA CHUNK-WISE EXECUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

On-policy distillation (OPD) trains student language models with teacher feedback on their own generations, aligning supervision with the states they visit. For long reasoning trajectories, however, conventional strict OPD execution incurs teacher-scoring waits and substantial computation and activation-memory costs during student training. Asynchronous execution improves utilization but introduces policy staleness that may degrade distillation quality, motivating a more efficient schedule that preserves strict on-policy property. We present ChunkOPD, which exploits prefix causality and student parameter consistency between rollout and training to reorganize both teacher scoring and student training around token chunks. Streaming Chunk Scoring scores committed prefixes before trajectories finish, overlapping teacher computation with rollout to shorten the scoring tail. Reverse Chunk Training reuses rollout key–value (KV) states to avoid an additional cache-population pass and recomputes chunks in reverse order to propagate gradients through the cached context while keeping activations local to each chunk. Across the evaluated workloads under matched physical GPU budgets, ChunkOPD achieves 1.11–1.98$\times$ speedups over synchronous OPD. In an end-to-end case study, ChunkOPD reduces time costs by 41% and achieves almost identical quality relative to synchronous OPD, whereas asynchronous OPD attains lower final accuracy.

1 INTRODUCTION

On-policy distillation (OPD) has demonstrated strong empirical performance across a growing body of work (Ko et al., 2025; Jia et al., 2026; Heo et al., 2026; Yang et al., 2026a). It trains a student language model on its own generations using token-level feedback from a teacher. By supervising the states the student actually visits, OPD reduces the mismatch between learning from fixed teacher-generated responses and generating under the student's own policy (Li et al., 2026b).

In a typical strict OPD implementation, the student generates trajectories, the teacher scores each completed trajectory, and the trainer computes student gradients from these scores (Kang et al., 2026). Rollout and training use the same student parameter version, with one optimizer update after gradients have accumulated over the full batch and before the next rollout batch begins. Two execution costs are particularly important for long trajectories. Waiting for completed trajectories can leave the teacher idle during rollout. The scoring work left after rollout forms a teacher tail on the critical path. The trainer then performs a differentiable forward pass and retains activations for backward, with an activation-memory footprint that grows with sequence length (Li et al., 2026a). Long reasoning trajectories increase scoring work and training computation and memory.

Prior work improves OPD efficiency by reducing rollout and supervision budgets or accelerating optimization (Zhang et al., 2026a;c; Yang et al., 2026b; Zhao et al., 2026; Tan et al., 2026; Gui et al., 2026; Xu et al., 2026; Cai et al., 2026), while systems optimizations reduce communication and loss-computation costs (Zhang et al., 2026b; Tilde Research, 2026). Even with these optimizations, teacher-scoring waits and the activation-memory demands of student training remain challenges for efficient strict OPD. Asynchronous OPD improves utilization by overlapping rollout with trainer updates, but the trainer may then learn from trajectories generated by an earlier student parameter version (Kang et al., 2026; Rang et al., 2026). This policy staleness introduces a distribution mismatch that can affect learning quality and stability, depending on the objective and how stale data

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.

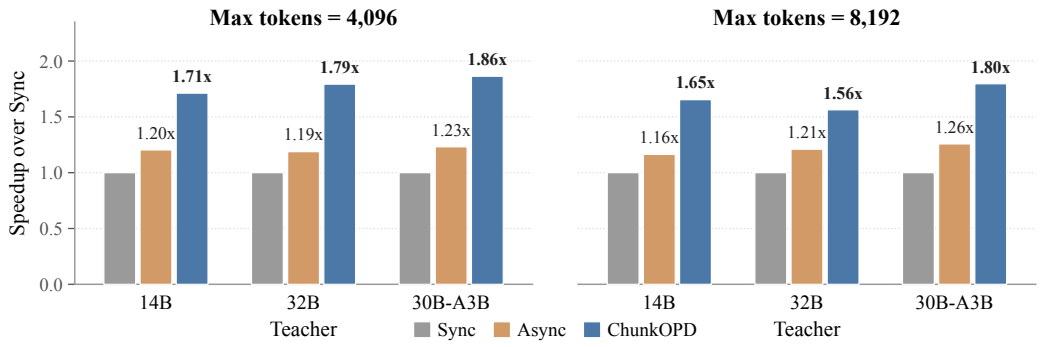


Figure 1: **Deployment speedups.** Qwen3-8B student, batch size 128, eight physical A100 GPUs. Speedup is relative to Sync with union placement for the same teacher and token cap.

are handled. We therefore target execution efficiency within strict on-policy updates, complementing these algorithmic and operation-level optimizations.

We introduce **ChunkOPD**, based on two observations about strict on-policy distillation. First, causal attention allows the teacher to score committed token prefixes without waiting for future tokens or trajectory completion. **Streaming Chunk Scoring** uses this property to overlap teacher computation with rollout and shorten the remaining scoring tail. Second, recomputing and differentiating chunks in reverse order reduces the activation workspace needed for training, but requires precomputed prefix key–value (KV) states because each chunk attends to its preceding context. In strict on-policy distillation, rollout already produces these states using the same student model and parameter version as training. **Reverse Chunk Training** combines rollout KV reuse with chunk-wise gradient reconstruction, keeping activations local to each chunk without an additional cache-population pass.

We evaluate ChunkOPD across student–teacher model pairs, dense and MoE teachers, four- and eight-GPU budgets, trajectory-length caps, and batch sizes. Under the same physical GPU budget, ChunkOPD achieves 1.11–1.98$\times$ speedups over synchronous OPD across all completed paired comparisons and even outperforms asynchronous OPD in most completed comparisons, while preserving strict on-policy updates (representative results in Figure 1). Component ablations and stage-time measurements show that Streaming Chunk Scoring shortens the post-rollout teacher-scoring tail, while Reverse Chunk Training reduces the student-training interval. In a 200-update end-to-end case study, ChunkOPD reduces training time by 41% relative to synchronous OPD with similar distillation quality, while asynchronous OPD obtains lower final accuracy.

Our contributions are: (1) **Streaming Chunk Scoring**: A schedule that exploits prefix causality to overlap teacher supervision with rollout within a strict on-policy update. (2) **Reverse Chunk Training**: A training design that reuses rollout KV to reconstruct causal gradients with chunk-local activations, without an additional cache-population pass. (3) **Implementation and Evaluation**: An open-source VeRL implementation (Sheng et al., 2025) with systematic evaluation of efficiency, component contributions, numerical consistency, and training quality.

2 Background and Related Work

2.1 On-policy distillation

Knowledge distillation trains a student to match a teacher through soft targets (Hinton et al., 2015), while sequence-level distillation trains on teacher-generated outputs (Kim & Rush, 2016). GKD and MiniLLM use student-generated responses to obtain supervision at student-visited contexts (Agarwal et al., 2024; Gu et al., 2024; Li et al., 2026b). Related work also studies adaptive divergence objectives (Wu et al., 2025) and alignment across tokenizers (Boizard et al., 2025; Zhang et al., 2024). Let θ_s denote the student parameters at step s and $\mathcal{D}_s = \{(x_i, y_i)\}_{i=1}^{B}$ a batch of prompt–response trajectories, where y_i is generated by $p_{\theta_s}(\cdot \mid x_i)$. At response position t, the frozen teacher

provides $q(\cdot \mid h_{i,t})$ for context $h_{i,t} = (x_i, y_{i,<t})$, giving the loss

$$\mathcal{L}_s^{\mathrm{OPD}}(\theta) = \frac{1}{N_s} \sum_{i,t} \mathcal{D}(q(\cdot \mid h_{i,t}), p_\theta(\cdot \mid h_{i,t})) \,, \tag{1}$$

where $\mathcal{D}$ is a chosen distillation discrepancy and N_s counts valid response positions. Trajectories and teacher targets are held fixed during differentiation.

Strict on-policy execution. A strict on-policy update uses the same student version θ_s for generation and gradient evaluation, then updates once after full-batch accumulation and publishes θ_{s+1} before the next rollout batch. This permits overlap within a batch while parameters remain fixed; it does not guarantee numerical agreement between inference and training backends.

2.2 Efficient distillation

Frameworks and operation-level efficiency. EasyDistill, TRL, and MS-SWIFT support distillation workflows (Wang et al., 2025; von Werra et al., 2020; Zhao et al., 2025). HybridFlow/verl and slime coordinate generation and optimization (Sheng et al., 2025; Zhu et al., 2025), while vLLM and SGLang provide efficient inference (Kwon et al., 2023; Zheng et al., 2024). KDFlow separates teacher inference from student training and transfers hidden states for student-side logit reconstruction (Zhang et al., 2026b); Nitrobrew fuses vocabulary projection with KL computation to reduce loss workspace (Tilde Research, 2026). These optimizations complement Streaming Chunk Scoring, which advances teacher computation into the rollout interval.

Rollout, supervision, and update budgets. DistiLLM reuses student-generated responses (Ko et al., 2024), while Lightning OPD precomputes teacher scores for offline training and its successor addresses cross-teacher style bias (Wu et al., 2026a;b). Prefix OPD, POPD/TOPD, and Adaptive FastOPD reduce computation through truncated or adaptive rollout horizons (Zhang et al., 2026a;c; Tan et al., 2026). Prune-OPD and PG-OPD use prefix information to adapt supervision or rollout allocation, while STRIDE combines early stopping with prefix-based restart for multi-turn OPD (Yang et al., 2026b; Zhao et al., 2026; Gui et al., 2026). TIP selects training positions, and EffOPD extrapolates parameter updates (Xu et al., 2026; Cai et al., 2026). ChunkOPD complements these budget choices by improving execution within each strict on-policy update.

Asynchronous and near-policy execution. AsyncOPD and Near-Policy decouple generation from training, allowing the learner to consume trajectories from earlier student versions (Kang et al., 2026; Rang et al., 2026). Staleness sensitivity depends on the objective and sample handling: AsyncOPD finds forward KL more robust than reverse KL with finite teacher-score caches, while Near-Policy uses selective filtering to stabilize optimization. ChunkOPD overlaps teacher scoring with rollout while retaining one student version until full-batch gradient accumulation completes.

Activation memory and rollout-state reuse. Activation checkpointing trades recomputation for memory (Chen et al., 2016), while FlashAttention reduces attention memory traffic (Dao et al., 2022). OOMB uses chunk-wise recomputation and KV-cache gradient propagation to limit activation storage (Li et al., 2026a). ChunkOPD's Reverse Chunk Training organizes gradient reconstruction around rollout KV from the same student parameter version, avoiding an additional cache-population pass. Each chunk still requires differentiable recomputation; reverse traversal releases its activations and frees suffix KV slots for overlapped transfer of the next microbatch's cache.

3 ChunkOPD

3.1 System Overview

Data flow. Figure 2 summarizes how ChunkOPD coordinates rollout, teacher scoring, and student training within a strict on-policy update. At step s, rollout workers generate responses with θ_s, publish committed token chunks, and retain student KV for training. Streaming Chunk Scoring processes these chunks incrementally, maintaining the teacher's own KV across submissions. A microbatch becomes ready for training once all its trajectories have terminated, its teacher targets

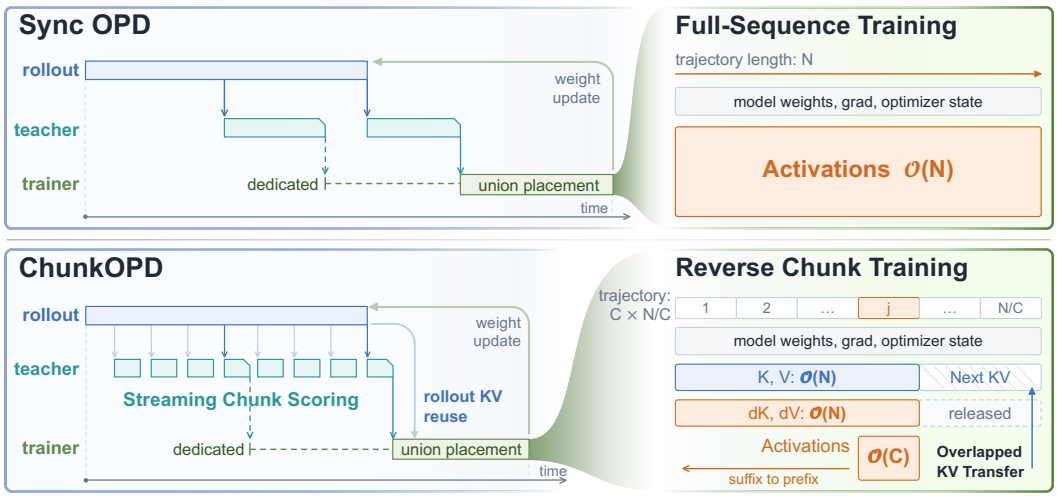


Figure 2: **ChunkOPD and synchronous OPD.** Left: execution schedules; dashed trainer segments indicate earlier starts under dedicated placement. Right: GPU memory layouts during training. Streaming Chunk Scoring overlaps teacher supervision with rollout, while Reverse Chunk Training reuses rollout KV for chunk-wise recomputation and backward.

are complete, and its student KV is available. Reverse Chunk Training then traverses the microbatch from suffix to prefix, reconstructing local activations and propagating gradients through the cached context. As backward completes each chunk, the trainer reuses its KV slots to load the next microbatch's cache while computation continues on the current prefix. Teacher-scoring and reverse-training chunks need not share boundaries.

Placement and update boundary. Rollout and teacher use disjoint GPU subsets in both placements. Under **union placement**, the trainer reuses their combined GPUs after inference drains. Under **dedicated placement**, it has a separate GPU pool and can start on a ready microbatch while inference continues, as indicated by the dashed segments in Figure 2. All microbatches use θ_s and accumulate gradients over the full batch before one optimizer update; the resulting θ_{s+1} is published before the next rollout batch begins.

3.2 Streaming Chunk Scoring

Streaming Chunk Scoring starts teacher scoring from committed rollout chunks before generation finishes. Each rollout request publishes monotonically growing, committed token ranges. The first teacher fragment contains the prompt and an initial response prefix; subsequent fragments extend the same teacher session. Causality makes the teacher distribution for an existing context independent of later response tokens. Retaining teacher KV between fragments therefore permits incremental scoring without repeatedly prefilling the complete prefix. A bounded session budget controls teacher-cache residency, and contiguous queued fragments may be coalesced before execution.

Supervision is indexed by *prediction position*. The final prompt position predicts the first response token, and a response position predicts its successor. Chunk boundaries do not reset this shift. The terminal row with no next-token target is excluded, as are padding and prompt-only loss positions. At EOS or length termination, the coordinator drains remaining fragments and verifies complete, nonoverlapping coverage of valid response targets. A training unit is ready only when all its trajectories terminate, their teacher targets are complete, and their student KV snapshots are sealed.

Streaming changes the teacher's start time, not the update frequency. It is also distinct from scoring a complete trajectory as soon as that trajectory finishes: the first teacher fragment can run *before* its trajectory reaches EOS. The corresponding ablation disables these pre-EOS fragments and scores each trajectory in a complete prefill after termination.

Latency effect. Let R and H denote rollout and teacher service times in a simplified two-stage model. Sequential inference takes $R + H$, whereas ideal overlap approaches $\max(R, H)$, plus pipeline fill, drain, and contention. Streaming reduces the teacher work left after rollout terminates; reducing total teacher busy time need not reduce step latency when rollout remains the bottleneck.

3.3 Reverse Chunk Training

Reverse Chunk Training uses rollout KV as causal context for chunk-wise recomputation and backward. This reuse avoids an additional cache-population pass before reverse traversal; each chunk still requires differentiable recomputation. Because rollout KV carries no autograd graph, recovering the full gradient requires propagating derivatives through the cached context. We traverse chunks from suffix to prefix, reconstructing local activations and accumulating gradients into every earlier cache block used by the current chunk.

Partition the prompt–response sequence into chunks $B_1, \ldots, B_m$, with layerwise cache entries $K_{\ell,j}, V_{\ell,j}$ for chunk j. Before processing j, maintain adjoints $A^K_{\ell,j}, A^V_{\ell,j}$ from already processed suffix chunks. Using the cached prefix as attention context, recompute the current chunk's keys $\widehat{K}_{\ell,j}$, values $\widehat{V}_{\ell,j}$, and unnormalized local loss $\mathcal{L}_j$. The required local vector–Jacobian product is represented by

$$\widetilde{\mathcal{L}}_j = \mathcal{L}_j + \sum_{\ell} \left[\left\langle \mathrm{sg}(A^K_{\ell,j}), \widehat{K}_{\ell,j} \right\rangle + \left\langle \mathrm{sg}(A^V_{\ell,j}), \widehat{V}_{\ell,j} \right\rangle \right], \tag{2}$$

where sg stops differentiation through the incoming adjoints. The implementation injects these terms directly in attention backward and accumulates outgoing adjoints in *every* attended prefix block, not only the immediately preceding chunk. Ragged trajectories are grouped by reverse depth with explicit validity masks, and loss normalization uses all valid response tokens in the full batch rather than separate chunk averages.

Gradient derivation. Consider fixed tokens and parameters θ_s under strictly causal decoder attention and exact arithmetic. Assume identical token positions, masks, model operations, and forward cache values between full and chunked execution, with no unmatched stochastic operations. The loss must decompose over valid response prediction positions with the shared full-batch denominator N_s. A shared policy version alone does not imply these conditions.

Let M_j collect all layerwise KV entries produced by chunk j and Y_j its outputs used by the loss. The chunk map retains all dependencies among layers and exposes the cached prefix as an input:

$$(Y_j, M_j) = F_j(\theta_s; M_{<j}, B_j), \qquad \mathcal{L}_s = \frac{1}{N_s} \sum_j \ell_j(Y_j). \tag{3}$$

Here $\ell_j(Y_j) = \mathcal{L}_j$ is the unnormalized loss in Eq. 2, and $\bar{M}_j$ collects adjoints from later uses of M_j on the same scale. The local vector–Jacobian product is

$$\left(\Delta g_j, \Delta \bar{M}_{<j}\right) = J^{\mathsf{T}}_{F_j} \left(\nabla_{Y_j} \ell_j, \bar{M}_j\right), \qquad \nabla_\theta \mathcal{L}_s = \frac{1}{N_s} \sum_j \Delta g_j. \tag{4}$$

Each backward accumulates parameter gradients and prefix adjoints without retaining the full-sequence autograd graph.

For the last chunk, $\bar{M}_m = 0$ because no later cache consumer exists. After differentiating chunks $m, \ldots, j+1$, $\bar{M}_j$ contains all gradient contributions from their uses of M_j. Differentiating F_j combines these contributions with its direct loss gradient and propagates both to every earlier cache block, establishing the same invariant for $j - 1$. Induction therefore recovers the full parameter gradient. Prompt-only chunks must still be visited: they have $\ell_j = 0$ but may receive nonzero $\bar{M}_j$. Detaching the prefix without retaining its adjoints breaks this invariant.

BF16 serving and training kernels can produce different forward values because their operations and prefill partitions differ. The derivation establishes chain-rule coverage under the stated conditions, not a numerical error bound; Section 4.2 and Appendix C measure the resulting discrepancies.

Memory and recomputation. For local batch size b, chunk size C, sequence length N, L layers, and hidden width d, local transformer activations scale approximately as $O(bCLd)$ and are released after each chunk's backward pass. KV and adjoint backing remains sequence-wide at $O(bNLh_{\mathrm{kv}}d_h)$, where h_{kv} and d_h are the KV-head count and head dimension (Figure 2, right). Model state, vocabulary workspace, and host-resident rollout caches add further memory costs. Smaller chunks reduce activation workspace but increase scheduling and kernel overhead; local batch size also affects this tradeoff.

Overlapped KV transfer. Consider consecutive microbatches u and $u+1$ within the same training batch at fixed student parameters. Once chunk B_j of u completes backward and accumulates its prefix gradients, all suffix chunks have been differentiated, and earlier chunks cannot attend to B_j. Its KV slots can therefore receive the corresponding cache range of $u+1$ while computation continues on the current prefix, whose states and adjoints remain intact. Progressively reusing these slots overlaps transfer with backward without allocating a second full cache (Figure 2, bottom right). The next microbatch starts only after its required loads finish; the initial load and any residual waits remain exposed. Appendix A.4 details the transfer protocol and its limitations.

4 Experiments

We evaluate four questions: how execution changes numerical results, whether ChunkOPD reduces update latency under the same physical GPU budget, which components produce the improvement, and whether ChunkOPD preserves training quality over successive updates.

4.1 Experimental setup

Experimental configuration. We adapt the distillation recipe of Li et al. (2026b). Prompts come from DAPO-Math-17K-Processed (Yu et al., 2025). The main matrix pairs Qwen3-4B and Qwen3-8B students with Qwen3-14B and Qwen3-32B teachers (Yang et al., 2025), using a batch size of 128 and total-token caps of 4096 and 8192. Additional experiments increase the batch to 256 or use a Qwen3-30B-A3B teacher. Performance runs use $4/8\times$ A100 GPUs on a single server. Computation and KV caches use BF16; the trainer uses FP32 gradient reductions, master parameters, and optimizer states. We report OPD update time as the elapsed time of the evaluated distillation update, including rollout, teacher scoring, student optimization, weight publication, KV handling, and scheduler lifecycle work. For short performance runs, we discard one warmup update and average the next three. Appendix B details GPU allocations, parallelism settings, and runtime options.

Baselines. We compare three baselines: synchronous OPD (**Sync**) as the standard strict on-policy reference, chunked forward (**CF**) as an alternative chunk-based training schedule, and asynchronous OPD (**Async**) as a throughput-oriented reference that allows policy staleness. Sync uses verl's native synchronous path. It scores each completed trajectory while unfinished rollouts continue, then trains on the full batch and publishes weights before the next rollout batch. CF retains Streaming Chunk Scoring and rollout KV reuse, but trains with forward chunks in sequence order, retaining the differentiable KV and activation graph needed for a full backward pass. It accumulates gradients over the full batch before updating weights, preserving strict on-policy training. Async uses verl's native asynchronous path to overlap rollout and training across policy versions. This baseline is separate from the AsyncOPD method of Kang et al. (2026). Appendix B.3 details the baselines.

Implementation and placement. We implement all methods in verl with vLLM inference, FSDP training, and the same top-k forward-KL objective (Kwon et al., 2023; Sheng et al., 2025). Sync and ChunkOPD (**Chunk**) use **union placement**: rollout and teacher occupy disjoint GPU subsets, and the trainer reuses their union after inference drains. CF and Async use **dedicated placement** with separate trainer, rollout, and teacher pools. In the matched dedicated-placement comparisons, CF and ChunkOPD share Streaming Chunk Scoring and rollout KV reuse, allowing comparison of their training schedules with the associated memory and batching plans. Appendix B lists the detailed configurations.

Table 1: **Dense-model OPD update time.** Mean seconds per update, batch 128; pairs denote Qwen3 student/teacher. All methods use the same physical GPU budget. Speedup is Sync/Chunk; OOM denotes a trainer-memory failure.

		4 GPUs					8 GPUs				
Model pair	**Cap**	**Sync**	**CF**	**Async**	**Chunk**	**Speedup**	**Sync**	**CF**	**Async**	**Chunk**	**Speedup**
4B/14B	**4096**	192.8	146.6	169.8	**112.9**	1.71×	117.0	94.7	109.6	**85.5**	1.37×
	8192	418.3	520.1	422.8	**278.4**	1.50×	242.8	274.9	211.2	**196.9**	1.23×
4B/32B	**4096**	189.2	245.9	158.1	**150.4**	1.26×	137.2	110.2	108.1	**92.1**	1.49×
	8192	418.6	645.1	432.8	**375.4**	1.11×	275.0	279.1	208.4	**208.3**	1.32×
8B/14B	**4096**	252.8	OOM	OOM	**163.0**	1.55×	221.7	137.3	184.2	**129.5**	1.71×
	8192	572.4	OOM	OOM	**416.1**	1.38×	464.3	407.9	399.0	**280.7**	1.65×
8B/32B	**4096**	307.2	OOM	OOM	**194.0**	1.58×	242.7	142.0	204.3	**135.4**	1.79×
	8192	650.8	OOM	OOM	**507.4**	1.28×	504.0	418.0	416.5	**322.5**	1.56×

4.2 Numerical Consistency

Although ChunkOPD is mathematically equivalent to Sync OPD under exact arithmetic, it replaces full-trajectory teacher prefill with chunked prefill using prefix KV and full-sequence training with chunk-wise execution. Because these different operator paths can introduce finite-precision discrepancies, we evaluate their numerical consistency using five disjoint fixed-trajectory groups comprising 640 trajectories and 2.54 million response tokens, with a Qwen3-0.6B/4B model pair. We compute each metric within a group and report an equally weighted average across groups.

Streaming Chunk Scoring achieves 99.554% top-1 agreement and 99.226% top-k overlap with scoring after each trajectory is complete. The probability-weighted absolute error in shared log-probabilities is 0.00813. Compared with KV from a standard forward pass, rollout KV has relative L2 differences of 0.785% for keys and 2.438% for values. Relative to native BF16 training with fixed teacher targets, ChunkOPD gradients have relative L2 differences of 4.312–4.700%. Yet despite these measurable finite-precision deviations from full-sequence training, ChunkOPD maintains strong overall gradient alignment, with five-group mean cosine similarities above 0.9988. Appendix C provides the detailed protocol and further analyses.

4.3 Update latency across models and GPU budgets

As shown in Table 1, ChunkOPD reduces OPD update time in all 16 settings, achieving speedups of 1.11–1.79× over Sync. Both ChunkOPD and Sync complete all eight four-GPU settings. In the four settings with an 8B student, both CF and Async fail because of trainer memory constraints. This feasibility gap reflects the different memory trade-offs: both methods reuse rollout KV, but CF retains activations and differentiable cross-chunk states until backward, whereas ChunkOPD reconstructs one chunk at a time during reverse training. ChunkOPD is also faster than Async in the completed settings except for one essentially tied case, while preserving strict on-policy updates. Async retains native student training, but uses dedicated pools, permits policy staleness, and may require renewed prefix prefill when weight updates interrupt unfinished rollouts. Across the 36 measured updates from the 12 completed Async main-matrix runs, the maximum consumed-trajectory staleness is one update; five updates contain a trajectory generated across two policy versions. Appendix B.3 reports the complete Async configuration and realized staleness statistics. Thus, the result is not only a lower average latency than that of Sync, but a broader feasible operating range than those of the alternative chunk-based and asynchronous schedules.

Both Sync and ChunkOPD become faster with more GPUs and slower with longer sequences or larger students, but differ in their sensitivity to these changes. For 8B students, moving from four to eight GPUs reduces ChunkOPD latency by 21–36%, compared with 12–23% for Sync. On eight GPUs, increasing student size from 4B to 8B raises latency by 1.43–1.55× for ChunkOPD versus 1.77–1.91× for Sync, showing that ChunkOPD accommodates larger students with a smaller relative latency increase. Conversely, increasing the token cap from 4096 to 8192 raises ChunkOPD latency by 2.17–2.62× versus 2.00–2.26× for Sync, narrowing the relative speedup in all eight paired set-

Table 2: **Placement, batch, and teacher variants.** Mean OPD seconds per update with a Qwen3-8B student; speedup is Sync/Chunk. Original: union placement, batch 128, Qwen3-32B teacher.

		Original			Union→Dedicated			Batch → 256			Teacher → MoE		
GPUs	Cap	Sync	Chunk	Speedup	Sync	Chunk	Speedup	Sync	Chunk	Speedup	Sync	Chunk	Speedup
4	4096	307.2	**194.0**	1.58×	359.1	**285.9**	1.26×	636.9	**355.7**	1.79×	280.4	**181.4**	1.55×
	8192	650.8	**507.4**	1.28×	OOM	818.6	—	1383.9	**882.8**	1.57×	744.9	**422.8**	1.76×
8	4096	242.7	**135.4**	1.79×	264.7	**137.9**	1.92×	407.3	**206.0**	1.98×	244.8	**131.3**	1.86×
	8192	504.0	**322.5**	1.56×	551.4	**411.4**	1.34×	826.0	**511.7**	1.61×	506.7	**282.0**	1.80×

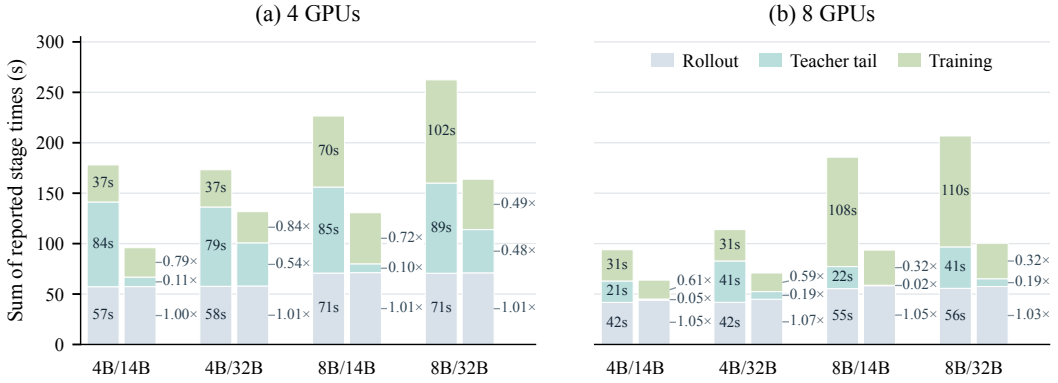


Figure 3: **Stage times for Sync (left) and ChunkOPD (right).** Union placement, batch 128, and Qwen3 student/teacher pairs at cap 4096. Stacks show rollout completion, the remaining teacher-scoring tail, and student training; weight publication and other step overheads are omitted.

tings. The resulting finding is that ChunkOPD is relatively more tolerant of student scaling, but its advantage narrows as the generated sequences become longer.

Table 2 extends the Qwen3-8B/32B comparison across placement, batch size, and teacher choice. Both ChunkOPD and Sync are faster under union than dedicated placement because union concentrates the physical GPU budget on each execution phase rather than statically partitioning it across concurrent roles. This reuse benefits ChunkOPD more under high load: at eight GPUs and cap 8192, union reduces ChunkOPD latency by 21.6%, compared with 8.6% for Sync; at four GPUs, dedicated Sync fails at this cap while ChunkOPD completes. At eight GPUs and cap 4096, however, union improves ChunkOPD by only 1.8%, suggesting that dedicated or asynchronous execution can be competitive when resources are ample; phase-wise pooling also helps explain why ChunkOPD can match or exceed Async under tighter budgets. Even without phase-wise pooling, ChunkOPD remains faster than Sync under dedicated placement on eight GPUs, with speedups of 1.92× and 1.34× at the two caps. Doubling the batch size increases ChunkOPD latency by only 1.52–1.83× and raises its speedup to as much as 1.98×, while the MoE teacher preserves speedups of 1.55–1.86×. Thus, ChunkOPD's advantage extends across the tested variants, with the largest placement benefit arising when the workload is high relative to the GPU budget.

4.4 Component Contributions

Under union placement, the update time can be primarily organized into the interval to rollout completion, the remaining teacher-scoring tail, and student training. Teacher scoring overlapped with generation is included in the rollout interval. The teacher-tail and trainer measurements characterize the two components within the selected execution configurations.

As shown in Figure 3, across the eight configurations, Streaming Chunk Scoring reduces the teacher tail to 0.02–0.54 times that of Sync, removing 46–98% of the post-rollout scoring wait and nearly hiding teacher scoring in the eight-GPU runs with the 14B teacher. The Reverse Chunk Training reduces the training interval to 0.32–0.84 times that of Sync, with the largest reductions for 8B students on eight GPUs, where training takes approximately 32% of Sync's time. Despite the re-

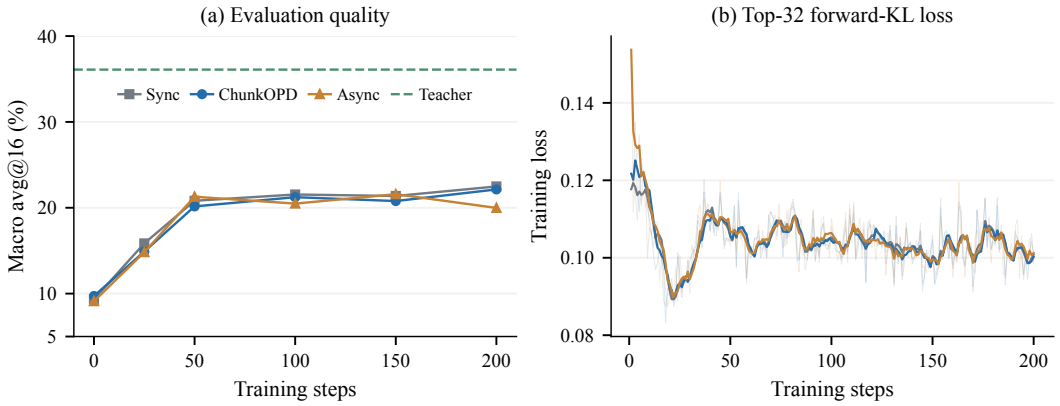


Figure 4: **Quality and loss over 200 steps.** (a) Macro avg@16 across AIME24, AIME25, and AMC23. (b) Top-32 forward-KL loss: per-step values (faint) and trailing 5-step means (solid).

Table 3: **200-update quality.** Audited avg@16 (%); fixed teacher reference; best student in bold.

Method	AIME24	AIME25	AIME mean	AMC23	Macro
Sync	12.71	**9.58**	**11.15**	**45.16**	**22.48**
Chunk	**12.92**	8.96	10.94	44.53	22.14
Async	10.21	7.08	8.65	42.66	19.98
Teacher (reference)	24.17	20.83	22.50	63.28	36.09

quired rollout KV export, the rollout interval remains 1.00–1.07 times that of Sync, an increase of only 0.1–3.0 seconds. Overall, the two components reduce complementary serial costs without materially extending rollout, showing the effectiveness of the key designs in ChunkOPD.

4.5 End-to-end training quality

Following the training recipe of Li et al. (2026b), we train a Qwen3-1.7B-Base student with a Qwen3-4B-Base-GRPO teacher on DAPO-Math-17K-Processed (Yu et al., 2025) for 200 updates, using four A100 GPUs per method. Appendix F provides the detailed settings and results.

ChunkOPD completes the measured 200-update OPD interval in 9.41 hours, 41% shorter than Sync (15.99 hours) and 27% shorter than Async (12.97 hours). Figure 4 shows that its loss trajectory closely tracks that of Sync, and Table 3 reports nearly identical final macro avg@16 scores: 22.14% for ChunkOPD and 22.48% for Sync. Async reaches 19.98% in this run. One possible explanation is policy staleness: asynchronous training may update the student using trajectories generated by earlier policy versions, which can reduce optimization stability. Quantifying its contribution to the quality gap requires a dedicated controlled study.

5 Conclusion

We presented ChunkOPD for efficient strict on-policy distillation. Streaming Chunk Scoring moves teacher supervision into the rollout interval, while Reverse Chunk Training reuses rollout KV to reconstruct causal gradients with chunk-local activations and no additional cache-population pass. Both components operate within a fixed student parameter version. Across the evaluated paired configurations, ChunkOPD achieves 1.11–1.98× speedups over synchronous OPD under the same physical GPU budget and outperforms asynchronous OPD in most completed comparisons. Ablations and stage-time measurements show that the two components reduce complementary costs: the teacher-scoring tail and the student-training interval. An end-to-end case study indicates that ChunkOPD achieves similar distillation quality over synchronous OPD with 41% less computational overhead. These results show that causal prefix scoring and reuse of rollout state can improve OPD execution efficiency while preserving strict on-policy updates.

Reproducibility Statement

Our implementation is available in an anonymous repository at https://anonymous.4open.science/r/ChunkOPD. The appendices provide detailed experimental settings, measurement and evaluation protocols, and numerical validation.

AI Use Statement

Generative AI produced most of the implementation code and assisted with manuscript preparation, literature lookup, consistency checks, and table and figure generation.

References

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=3zKtaqxLhW.

Nicolas Boizard, Kevin El Haddad, Céline Hudelot, and Pierre Colombo. Towards Cross-Tokenizer Distillation: the Universal Logit Distillation Loss for LLMs. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=bwRxXiGO9A.

Yuchen Cai, Ding Cao, Liang Lin, Chunxi Luo, Xin Xu, Kai Yang, Weijie Liu, Saiyong Yang, Tianxiang Zhao, Guangzhong Sun, Guiquan Liu, and Junfeng Fang. Learning to Foresee: Unveiling the Unlocking Efficiency of On-Policy Distillation, 2026. URL https://arxiv.org/abs/2605.11739.

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, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359. Curran Associates, Inc., 2022. doi: 10.52202/068431-1189. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge Distillation of Large Language Models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=5h0qf7IBZZ.

Zhiyu Gui, Kexin Huang, Jia Guo, Junkang Wu, Zihao Wang, Zhiqiang Zhang, Jun Zhou, Jiancan Wu, and Xiang Wang. Know When to Stop, Where to Restart: Accelerating Multi-Turn Agentic On-Policy Distillation, 2026. URL https://arxiv.org/abs/2609.14636.

Byeongho Heo, Jaehui Hwang, Sangdoo Yun, and Dongyoon Han. On-Policy Delta Distillation. arXiv preprint arXiv:2607.15161, 2026. URL https://arxiv.org/abs/2607.15161.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. arXiv preprint arXiv:1503.02531, 2015. URL https://arxiv.org/abs/1503.02531.

Nan Jia, Haojin Yang, Xing Ma, Jiesong Lian, Shuailiang Zhang, Weipeng Zhang, Ke Zeng, Xunliang Cai, and Zequn Sun. Asymmetric On-Policy Distillation: Bridging Exploitation and Imitation at the Token Level. arXiv preprint arXiv:2605.06387, 2026. URL https://arxiv.org/abs/2605.06387.

Wonjun Kang, Kevin Galim, Seunghyuk Oh, Minjun Kang, Sanghyun Park, Donghoon Kim, Minjae Lee, Minseo Kim, Rishabh Tiwari, Yuchen Zeng, Hyung Il Koo, and Kangwook Lee. AsyncOPD: How Stale Can On-Policy Distillation Be? In *Decision-Making from Offline Datasets to Online Adaptation: Black-Box Optimization to Reinforcement Learning*, 2026. URL https://openreview.net/forum?id=U0kMQDWoaQ. ICML 2026 DEMO Workshop.

Yoon Kim and Alexander M. Rush. Sequence-Level Knowledge Distillation. In Jian Su, Kevin Duh, and Xavier Carreras (eds.), *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1317–1327, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1139. URL https://aclanthology.org/D16-1139/.

Jongwoo Ko, Sungnyun Kim, Tianyi Chen, and Se-Young Yun. DistiLLM: Towards Streamlined Distillation for Large Language Models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24872–24895. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/ko24c.html.

Jongwoo Ko, Tianyi Chen, Sungnyun Kim, Tianyu Ding, Luming Liang, Ilya Zharkov, and Se-Young Yun. DistiLLM-2: A Contrastive Approach Boosts the Distillation of LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 31044–31062. PMLR, 2025. URL https://proceedings.mlr.press/v267/ko25a.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626. ACM, October 2023. doi: 10.1145/3600006.3613165. URL http://dx.doi.org/10.1145/3600006.3613165.

Wenhao Li, Daohai Yu, Gen Luo, Yuxin Zhang, Yifan Wu, Jiaxin Liu, Ziyang Gong, Zimu Liao, Fei Chao, and Rongrong Ji. Out of the Memory Barrier: A Highly Memory-Efficient Training System for LLMs with Million-Token Contexts. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=dSa3ImCQr7.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan ang Gao, Wenkai Yang, Zhiyuan Liu, and Ning Ding. Rethinking On-Policy Distillation of Large Language Models: Phenomenology, Mechanism, and Recipe. In *ICML 2026 Workshop on Foundations of Deep Generative Models: Understanding Memorization, Generalization, and Reasoning*, 2026b. URL https://openreview.net/forum?id=aaDDXXDGXB.

Miao Rang, Zhenni Bi, Hang Zhou, Kai Han, Xuechun Wang, An Xiao, Xinghao Chen, Yunhe Wang, and Hanting Chen. Near-Policy: Accelerating On-Policy Distillation via Asynchronous Generation and Selective Packing. arXiv preprint arXiv:2605.05940, 2026. URL https://arxiv.org/abs/2605.05940.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 1279–1297. ACM, March 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.

Qian Tan, Huaifei Liang, Xuanyu Zhu, Lei Jiang, and Yuqiang Li. Adaptive FastOPD: Progress-Aware Rollout Horizon Expansion for Efficient On-Policy Distillation, 2026. URL https://arxiv.org/abs/2607.29494.

Tilde Research. Nitrobrew. Software repository, 2026. URL https://github.com/tilde-research/nitrobrew-release. Accessed September 26, 2026.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning. Software repository, 2020. URL https://github.com/huggingface/trl.

Chengyu Wang, Junbing Yan, Wenrui Cai, Yuanhao Yue, and Jun Huang. EasyDistill: A Comprehensive Toolkit for Effective Knowledge Distillation of Large Language Models. In Ivan Habernal, Peter Schulam, and Jörg Tiedemann (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 787–795, Suzhou,

China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-334-0. doi: 10.18653/v1/2025.emnlp-demos.60. URL https://aclanthology.org/2025.emnlp-demos.60/.

Taiqiang Wu, Chaofan Tao, Jiahao Wang, Runming Yang, Zhe Zhao, and Ngai Wong. Rethinking Kullback-Leibler Divergence in Knowledge Distillation for Large Language Models. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (eds.), *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 5737–5755, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.383/.

Yecheng Wu, Song Han, and Han Cai. Lightning OPD: Efficient Post-Training for Large Reasoning Models with Offline On-Policy Distillation. arXiv preprint arXiv:2604.13010, 2026a. URL https://arxiv.org/abs/2604.13010.

Yecheng Wu, Song Han, and Han Cai. Lightning OPD 2.0: Mitigating Style Bias in Cross-Teacher On-Policy Distillation for Large Reasoning Models. arXiv preprint arXiv:2607.28449, 2026b. URL https://arxiv.org/abs/2607.28449.

Yuanda Xu, Hejian Sang, Zhengze Zhou, Ran He, Zhipeng Wang, and Alborz Geramifard. TIP: Token Importance in On-Policy Distillation, 2026. URL https://arxiv.org/abs/2604.14084.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report. arXiv preprint arXiv:2505.09388, 2025. URL https://arxiv.org/abs/2505.09388.

Run Yang, Runpeng Dai, Jie Sun, Jielei Zhang, Fan Zhou, Hongtu Zhu, Peiyi Li, and Longwen Gao. Influence-Directed Distillation: Solving the Diversity Bottleneck in Sampled-Token On-Policy Distillation. arXiv preprint arXiv:2608.29846, 2026a. URL https://arxiv.org/abs/2608.29846.

Zhicheng Yang, Zhijiang Guo, Yifan Song, Minrui Xu, Yongxin Wang, Yiwei Wang, Xiaodan Liang, and Jing Tang. Prune-OPD: Efficient and Reliable On-Policy Distillation for Long-Horizon Reasoning, 2026b. URL https://arxiv.org/abs/2605.07804.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2a36EMSSTp.

Dongxu Zhang, Zhichao Yang, Sepehr Janghorbani, Jun Han, Andrew Ressler, II, Qian Qian, Gregory D Lyng, Sanjit Singh Batra, and Robert E. Tillman. Fast and Effective On-Policy Distillation from Reasoning Prefixes. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 25553–25569, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1276. URL https://aclanthology.org/2026.findings-acl.1276/.

Songming Zhang, Xue Zhang, Zengkui Sun, Yufeng Chen, and Jinan Xu. Dual-Space Knowledge Distillation for Large Language Models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen

(eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 18164–18181, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.1010. URL https://aclanthology.org/2024.emnlp-main.1010/.

Songming Zhang, Xue Zhang, Tong Zhang, Bojie Hu, Yufeng Chen, and Jinan Xu. KDFlow: A User-Friendly and Efficient Knowledge Distillation Framework for Large Language Models. arXiv preprint arXiv:2603.01875, 2026b. URL https://arxiv.org/abs/2603.01875.

Yaocheng Zhang, Jiajun Chai, Yuqian Fu, Songjun Tu, Xiaohan Wang, Wei Lin, Guojun Yin, Qichao Zhang, Yuanheng Zhu, and Dongbin Zhao. Are Full Rollouts Necessary for On-Policy Distillation? arXiv preprint arXiv:2605.31490, 2026c. URL https://arxiv.org/abs/2605.31490.

Qingfei Zhao, Huan Song, Shuyu Tian, Jiawei Shao, and Xuelong Li. Prefix-Guided On-Policy Distillation: Mining Golden Trajectories from Rollouts, 2026. URL https://arxiv.org/abs/2606.21994.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. SWIFT: A Scalable Lightweight Infrastructure for Fine-Tuning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(28):29733–29735, April 2025. ISSN 2159-5399. doi: 10.1609/aaai.v39i28.35383. URL http://dx.doi.org/10.1609/aaai.v39i28.35383.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL Scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

A OBJECTIVE AND EXECUTION DETAILS

A ChunkOPD update computes teacher targets and student gradients for a fixed batch of trajectories. The loss specifies the supervised prediction positions, while teacher sessions, reverse training, and cache transfers determine the execution order. Streaming Chunk Scoring and Reverse Chunk Training use the same trajectories but choose their chunk boundaries independently.

A.1 DISTILLATION OBJECTIVE AND PREDICTION-POSITION ALIGNMENT

Let $z_i = (x_i, y_i)$ concatenate the prompt and generated response of trajectory i. Let $\mathcal{I}_i$ contain the positions whose next token belongs to the retained response, and let $N_s = \sum_i |\mathcal{I}_i|$. The teacher selects its top-k vocabulary entries $S_{i,t}$ at each supervised context $h_{i,t}$. All compared schedules use the truncated forward-KL objective

$$\mathcal{L}_s(\theta) = \frac{1}{N_s} \sum_i \sum_{t \in \mathcal{I}_i} \sum_{v \in S_{i,t}} q(v \mid h_{i,t}) \big[\log q(v \mid h_{i,t}) - \log p_\theta(v \mid h_{i,t}) \big]. \tag{5}$$

The teacher is frozen, and both trajectories and teacher targets are held fixed when computing gradients. Probabilities use full-vocabulary normalization; the selected entries are not renormalized within $S_{i,t}$. This truncation differs from computing KL after normalizing both distributions over the top-k set. Full-vocabulary projection and normalization are still required to evaluate the selected student log-probabilities. The performance configurations retain $k = 32$, use no task-reward or policy-gradient term, and do not apply a log-probability or loss clamp.

Prediction positions. A logit row predicts the token following its input position. For the sequence $(x_1, x_2, y_1, y_2, y_3, \mathrm{EOS})$, the row at x_2 supervises y_1 and the row at y_3 supervises EOS. The final EOS row has no retained successor and contributes no loss. If generation stops at the length cap, the last retained response token is still supervised by its predecessor, but no additional target is created after it. Table 4 shows that splitting the input after y_1 does not alter this indexing.

Table 4: Prediction-position alignment for a two-token prompt and a response ending in EOS. The split marks a chunk boundary; supervision uses the original sequence positions.

Input position	Input token	Chunk	Next token	Response loss	Role
1	x_1	1	x_2	No	Prompt context
2	x_2	1	y_1	Yes	First response target
3	y_1	1	y_2	Yes	Cross-boundary target
4	y_2	2	y_3	Yes	Response target
5	y_3	2	EOS	Yes	Termination target
6	EOS	2	—	No	No retained successor

Prompt positions without a response target and padding rows have zero direct loss. They can still affect later response predictions through attention, so their hidden states and KV can receive gradients even when their direct loss is zero. For a chunk that crosses the boundary between prompt and response, the trainer applies the loss mask at each position. Teacher targets and student predictions share absolute prediction indices regardless of their chunk sizes.

A.2 STREAMING CHUNK SCORING

Each rollout request has an identifier within the current policy batch and publishes committed token ranges in sequence order. The teacher initializes a session with the prompt and first response fragment. Subsequent fragments extend that session's cached prefix, so the teacher does not prefill the complete trajectory for every fragment. The coordinator may merge adjacent queued fragments before submission without changing their prediction indices. It limits the number of resident sessions and their reserved KV capacity to bound teacher memory use. Requests waiting for teacher admission keep their pending fragments until a session can accept them.

When generation reaches EOS or the length cap, the coordinator submits the remaining fragment and drains the teacher session. Teacher scoring for the request completes once every valid response

target has been collected exactly once. The final fragment can be shorter than the configured chunk size. The terminal prediction row remains masked even inside an otherwise valid fragment. The coordinator releases teacher KV after collecting the request's targets; student rollout KV remains available for training, as described below.

Streaming Chunk Scoring can start before a trajectory terminates. The *w/o Streaming Chunk Scoring* ablation instead submits a complete prefill after each trajectory finishes, using the same response targets. Both schedules keep the policy fixed throughout the update.

A.3 Reverse Chunk Training

Reverse Chunk Training uses the sealed rollout cache as causal context for local differentiable recomputation. For each training microbatch, it maintains parameter-gradient accumulators and layerwise adjoints for the live prefix KV. The cached values have no autograd graph, so each recomputed chunk must account for its direct loss and incoming gradients from later chunks. Equations 2 and 4 define the local vector-Jacobian product.

Figure 5 gives the reverse traversal in pseudocode. Traversing chunks in reverse accumulates all suffix contributions to a prefix before differentiating it. Attention backward writes to every earlier cache block used by the current chunk; restricting propagation to the immediately preceding block would omit causal dependencies. A prompt-only chunk has zero local loss but still receives and propagates suffix adjoints. Once a chunk's backward pass finishes, no remaining computation in the current microbatch needs its KV, so the next microbatch can reuse those slots.

Reverse Chunk Training for microbatch u

Inputs: fixed parameters θ_s, tokens and masks, teacher targets, sealed rollout KV, and optional next microbatch $u + 1$.

1. Load the initial cache and wait until the required pages are ready. Initialize the microbatch's KV adjoints to zero.
2. Visit valid reverse depths from the final chunk toward the prompt. For each chunk j:
   (a) Recompute its local activations using the cached prefix, absolute positions, and causal mask.
   (b) Form the unnormalized local response loss and inject the incoming layerwise A_j^K, A_j^V.
   (c) Backpropagate into the shared parameter-gradient accumulator and all earlier KV-adjoint slots.
   (d) Record backward completion, retire the consumed suffix state, and allow its KV slots to receive pages from $u + 1$.
3. Release the completed microbatch's remaining state. Start $u + 1$ only when its required cache loads have completed. Keep parameters unchanged until the full policy batch is accumulated.

Figure 5: Reverse training pseudocode. Gradient propagation, cache retirement, and next-microbatch loading are parts of the same schedule.

Ragged trajectories and normalization. Trajectories can end at different lengths. The trainer groups chunks by reverse depth, masks invalid positions, and trims padding where the plan permits. A short final chunk contributes loss only at its valid positions. Distributed workers coordinate backward calls so that parameter-gradient synchronization remains compatible across ranks. The trainer normalizes accumulated gradients by the full batch's valid response-token count, using a common denominator across chunks and microbatches. After all training units finish, it completes any required gradient synchronization, clips the gradients, and applies one optimizer update.

Conditions for graph equivalence. The derivation in Section 3.3 assumes identical forward values, positions, masks, model operations, and stochastic choices at fixed parameters. Under these conditions, local recomputation and KV-adjoint propagation follow the same chain-rule paths as the full causal graph. Even at the same policy version, inference and training engines can produce different forward values. Appendix C measures these finite-precision differences.

A.4 KV LIFECYCLE AND OVERLAPPED TRANSFER

A rollout cache first belongs to the inference request that produces it. When the trajectory terminates, the rollout worker exports the required states through pinned host staging and seals them for the trainer. The training unit becomes eligible only after all of its trajectories have terminated, all targets are complete, and all snapshots are sealed. The teacher and student caches belong to different models and are never substituted for one another.

Within the trainer, a slot progresses through *active*, *backward complete*, *loading next*, and *next ready* states. The trainer records a CUDA event after backward and prefix-adjoint accumulation to mark when the slot can be overwritten. The copy stream waits for this event, loads the corresponding KV range of the next microbatch, and records a load-completion event. The compute stream can continue backward on the remaining current prefix while these pages are transferred. The trainer starts the next microbatch only after all required cache loads finish. Unused rows and suffix pages of ragged microbatches may be available earlier because they have no current consumer.

As the current cache shrinks, the next microbatch's cache fills the slots it releases. The implementation also supports a separate inactive KV buffer when the memory plan permits it. Appendix B reports the slot allocation and local batch selected by the training plan. Both paths use fixed parameters and cache states from the current policy version.

Exposed work. The first microbatch must wait for its initial cache load before backward can begin. Later microbatches also wait if their loads outlast the current computation. A chunk spanning the full slot leaves no earlier prefix computation with which to overlap incremental suffix loading. The chunk layout and workload therefore determine how much transfer can overlap computation. Copy durations can include transfers that overlap computation, so adding them to compute time can double-count elapsed time.

A.5 FULL-UPDATE SCHEDULE AND POLICY CONSISTENCY

Figure 6 combines the teacher and trainer protocols. Under union placement, inference pools drain and release their resources before the trainer uses their union. Under dedicated placement, a completed training unit may enter the trainer while other trajectories are still generating or being scored. Both schedules accumulate the full batch at θ_s and publish θ_{s+1} before the next strict on-policy rollout batch.

One strict on-policy update

Inputs: student version θ_s, frozen teacher, prompt batch, and placement plan.

1. Fix θ_s on the rollout and training workers and clear the batch-gradient accumulator.
2. Generate responses and publish committed fragments to persistent teacher sessions. Retain student KV for each trajectory.
3. At trajectory termination, drain its teacher fragments, verify target coverage, and seal its exported student KV.
4. Mark a training unit ready after every member satisfies the completion conditions. Launch ready units when the placement plan provides trainer resources.
5. Apply Reverse Chunk Training to each unit, accumulating gradients without changing θ_s. Overlap next-microbatch KV loads within the trainer where possible.
6. After the full batch completes, finalize normalization and synchronization, clip gradients, and apply one optimizer update.
7. Publish θ_{s+1} and complete the publication barrier before starting the next rollout batch.

Figure 6: Full-update pseudocode. Teacher chunks and reverse-training chunks are independent execution partitions of the same trajectories.

B EXPERIMENTAL SETUP AND MEASUREMENT PROTOCOL

This section specifies the workloads, hardware, and runtime settings used in our numerical, performance, and training-quality experiments. It also defines the timing windows and statistics reported in the results.

B.1 MODELS, DATASETS, AND WORKLOADS

Performance prompts come from DAPO-Math-17K-Processed. For short performance runs, all methods use the same dataset snapshot and data-loader shuffle seed of 1. This gives them the same source prompt order, although Async's queue can change the order in which it consumes completed prompt groups. We report the configured total-token budget as the cap, using 4096 and 8192 tokens in the short performance experiments. Short performance runs use greedy decoding (`do_sample=False`) to generate one response per prompt. Actual trajectory lengths depend on prompt length and generation termination. The numerical audit uses fixed greedy trajectories; the end-to-end experiment uses its own prompt schedule and evaluation settings in Appendix F. All comparisons share a tokenizer within each student-teacher pair and the top-32 objective in Appendix A.1.

Prompt-length distribution. We count prompt tokens after applying the Qwen3 chat template, including the assistant generation prefix and excluding padding; the 4B and 8B student tokenizers give identical lengths on this dataset. The dataset snapshot contains 14,116 prompts, with mean/median lengths of 100.1/87 tokens and a maximum of 1496 tokens. Figure 7 compares the full dataset with the fixed-seed samples used by the strict schedules in the three measured updates, excluding the warmup update. At batch sizes 128 and 256, these samples contain 384 and 768 prompts, with mean lengths of 98.8 and 101.2 tokens and maxima of 524 and 850 tokens, respectively. No prompt is truncated in these measured windows or their corresponding warmup batches. The short-run configuration disables dataset-level overlength filtering. The plotted samples cover these fixed input windows; Async may consume or prefetch different groups through its queue.

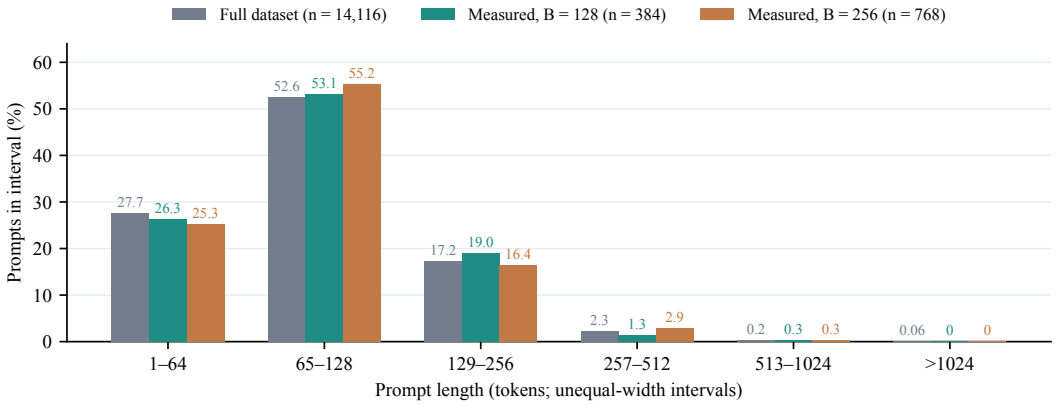


Figure 7: Prompt-length distributions in the source dataset and fixed-seed performance samples. Bars show the percentage of prompts in each labeled token interval. The intervals have unequal widths, so bar heights represent probability mass rather than density. Measured samples exclude one warmup update and include the following three updates.

B.2 HARDWARE AND EXECUTION CONFIGURATIONS

Each performance run uses NVIDIA A100-SXM4 80GB GPUs on a single server with two 32-core Intel Xeon Platinum 8358P CPUs and approximately 1 TiB of host memory. GPUs 0 to 3 and 4 to 7 occupy separate NUMA domains. The recorded topology has twelve bonded NVLinks within each of the pairs (0,1), (2,3), (4,5), and (6,7); communication between pairs uses PCIe and, across domains, the CPU interconnect. Student and teacher computation uses BF16, as do inference caches

and the exported rollout KV. The FSDP training path uses FP32 gradient reductions and retains FP32 master parameters and AdamW states. The student uses FlashAttention 2 in the BF16 training paths.

Role allocations. We write trainer/rollout/teacher GPU counts as T/R/H. Union placement assigns separate rollout and teacher pools during inference, then makes their union available to the trainer. Dedicated placement assigns three disjoint pools. Table 5 lists the GPU assignments. Under union placement, T reuses the R and H GPUs at a later stage, so T/R/H does not describe three disjoint allocations. Rollout tensor parallelism is one in the main performance configurations. Teacher tensor parallelism is given separately in the table; an H-GPU teacher pool contains H/TP inference replicas. The FSDP world size equals T, with one training rank per GPU and no additional student tensor or sequence parallelism.

Allocation selection. Each method uses a runtime configuration selected for its memory and overlap requirements under the same physical GPU budget. We chose role splits through memory checks and short pilot runs, comparing update latency and stage balance on representative workloads. Union placement balances the inference pools before reusing all GPUs for training; dedicated placement reserves trainer capacity for model state and activations. For the 4B student at cap 8192 on eight GPUs, T2/R4/H2 assigns more GPUs to the longer rollout workload and fewer to training. We selected teacher tensor parallelism alongside these allocations to fit the model and KV cache in memory. The comparisons therefore include the effects of these resource allocations and runtime settings.

Table 5: GPU role allocations for performance experiments. The eight-GPU 4B/cap-8192 entry applies to both dense teachers. For eight-GPU dense runs, TP is 1 for the 14B teacher and 2 for the 32B teacher.

Deployment	Physical GPUs	T/R/H	Teacher TP
Sync/Chunk union	4	4/2/2	2
Sync/Chunk union	8	8/4/4	1/2
CF/Async dedicated	4	2/1/1	1
CF/Async dedicated, standard	8	4/2/2	1/2
CF/Async dedicated, 4B, cap 8192	8	2/4/2	1/2
8B/32B dedicated controls	4	2/1/1	1
8B/32B dedicated controls	8	4/2/2	2
MoE dedicated	8	4/2/2	2

Inference settings. The settings below apply to the dense main matrix. The batch-size extension and dedicated controls use the same execution paths with capacity settings adjusted for each workload. Memory fractions denote vLLM `gpu_memory_utilization`, and sequence limits denote `max_num_seqs` per inference replica. We test increasing memory fractions, including 0.5, 0.75, and 0.85, and use the largest that fits the workload, lowering the fraction after an out-of-memory failure. Full-sequence teacher logit computation is the main capacity constraint for Sync and Async, so their teacher fractions can be lower than those of chunked paths. ChunkOPD uses memory fractions of 0.85 for both inference roles and rollout/teacher sequence limits of 64/128 on four GPUs and 32/128 on eight GPUs. For CF, rollout uses a memory fraction of 0.85 and a sequence limit of 32 to 128; the teacher uses 0.8 to 0.9 and 32. Sync uses rollout memory fractions of 0.5 to 0.7 and 256 sequences, with a teacher memory fraction of 0.5 and 1024 sequences in the main matrix. Async uses 0.5 and 256 for rollout, and 0.5 and 1024 for the teacher, except for a teacher fraction of 0.9 in the selected four-GPU 4B/32B runs. For ChunkOPD and CF, `max_num_batched_tokens` equals the trajectory cap (4096 or 8192) for both inference roles; Sync and Async use 8192 for both roles. These are per-engine scheduling limits, not the total token count of a policy batch. For ChunkOPD, the teacher engine allows at most 128 sequences, but the planner can impose a lower active-trajectory limit based on teacher KV capacity and the reservation per trajectory. When that limit is below 128, KV capacity determines admission. The batch-256 ChunkOPD extension allows rollout sequence limits up to 128; dedicated Sync controls use teacher memory fractions up to 0.9. The resolved configurations retain the exact capacity settings for each run.

Student training and sharding. All four methods enable activation checkpointing and disable activation offload. They use FSDP over the trainer pool, with FP32 master parameters and AdamW states, BF16 forward/backward computation, and FP32 gradient reduction. FSDP uses `FULL_SHARD` where the memory plan and execution path permit it, with `SHARD_GRAD_OP` retaining gathered parameters through backward while still sharding training state across ranks. ChunkOPD and CF use `SHARD_GRAD_OP` to retain gathered parameters across their repeated chunk computations and avoid repeated parameter gathering. For Sync and Async, pilots found similar performance with the two sharding modes, so we use `FULL_SHARD` to retain memory headroom for their native training paths and batch settings. Union-placement Sync and ChunkOPD offload parameters and optimizer states to release trainer memory during inference. Dedicated configurations keep both on their assigned GPUs so inference and training can overlap. Async uses dedicated placement because its rollout and training overlap spans policy versions; Table 18 provides dedicated Sync and ChunkOPD controls. ChunkOPD and CF use custom chunk-aware operators, drawing on the implementation approach of OOMB (Li et al., 2026a), to support their nonstandard KV handling and differentiation paths. These training paths enable Liger kernels as part of their operator integration. Sync and Async use the native full-sequence operators with Liger disabled. The Liger setting reflects these differences in operator implementation rather than a shared optimization applied to otherwise identical training paths.

Batching and gradient accumulation. The main matrix uses one training epoch and one optimizer update per batch of 128 trajectories, with gradients accumulated across its training microbatches. All four configurations enable dynamic batching, but the native token packer and the chunk planners determine their training units differently. Native training packs complete trajectories within a per-rank token budget. Chunk training plans for the local batch, active chunk, cached context, and retained gradient state together. For Sync and Async, `ppo_max_token_len_per_gpu` is 16384 for the 4B student and 8192 for the 8B student; the number of microbatches therefore depends on realized trajectory lengths. ChunkOPD sets this field to the trajectory cap and uses the realized reverse plan below to select its local batch and chunk size. CF sets it to zero to request automatic memory planning, yielding a per-rank token budget of 32768 for the 4B student and 8192 for completed 8B runs in the main matrix. We enable `no_sync` whenever the resolved memory plan and training path support it, deferring parameter-gradient synchronization until the full batch has been accumulated. With FSDP, deferring synchronization reduces collectives but retains unsharded accumulated gradients. Whether these gradients fit in memory depends on the sharding mode and live activation and KV state. ChunkOPD and CF enable `no_sync` accumulation in the reported configurations. Async disables `no_sync`; Sync also disables it except in the four-GPU 8B main-matrix runs, which enable it. Without `no_sync`, microbatch gradients are reduced during accumulation, but the optimizer still steps only once per batch. These choices change the memory and communication schedule while preserving the full-batch update boundary and token-normalized objective. For ChunkOPD's local batch $b = 2$, a batch of 128 requires $128/(2T)$ reverse microbatches per rank, each containing multiple chunk backward calls when needed.

ChunkOPD training plans. The main matrix uses teacher chunks of $C_H = 1024$ tokens. The reverse planner chooses the chunk size C_R and local batch from available memory, subject to the configured limits. The chosen chunk size can be smaller than its configured upper bound. Table 6 lists the resulting plans, all with local batch $b = 2$. For the 4B student, $C_R = 2048$ except for the eight-GPU, cap-4096 case, which uses 4096; the 8B student uses $C_R = 1024$. KV slots use 64-token pages and cover the trajectory cap, with 2.25 to 4.50 GiB of recorded backing per training rank as listed in the table. The next microbatch reuses retired suffix slots; the trainer does not retain a differentiable graph for the complete trajectory. Planning accounts for model state, cached KV, adjoints, active-chunk activations, and loss workspace. The sensitivity experiments explicitly vary teacher or reverse chunk size.

B.3 Baseline implementations

Synchronous execution. Sync starts teacher scoring when a trajectory terminates, while unfinished requests continue generating. The trainer starts after all trajectories and teacher targets for the batch are available. Weight publication follows the training update and completes before the next rollout batch. The Sync inference engines enable `free_cache_engine`. At the inference-

Table 6: Realized ChunkOPD training plans in the main matrix, batch 128. C_H: teacher chunk size; C_R: maximum realized reverse chunk size; b: planned local batch. KV-slot backing is the recorded trainer allocation, not total device memory.

GPUs	Pair	Cap	C_H	C_R	b	KV slots (GiB)
4	4B/14B	4096	1024	2048	2	2.25
4	4B/14B	8192	1024	2048	2	4.50
4	4B/32B	4096	1024	2048	2	2.25
4	4B/32B	8192	1024	2048	2	4.50
4	8B/14B	4096	1024	1024	2	2.25
4	8B/14B	8192	1024	1024	2	4.50
4	8B/32B	4096	1024	1024	2	2.25
4	8B/32B	8192	1024	1024	2	4.50
8	4B/14B	4096	1024	4096	2	3.38
8	4B/14B	8192	1024	2048	2	4.50
8	4B/32B	4096	1024	4096	2	3.38
8	4B/32B	8192	1024	2048	2	4.50
8	8B/14B	4096	1024	1024	2	2.25
8	8B/14B	8192	1024	1024	2	4.50
8	8B/32B	4096	1024	1024	2	2.25
8	8B/32B	8192	1024	1024	2	4.50

Table 7: Execution behavior of the four compared methods. Strict denotes one student parameter version for generation and gradient evaluation within a full training batch.

Method	Teacher submission	Student-training path	Rollout KV	Strict
Sync	Complete trajectory	Native full-sequence training after the batch is ready	Not reused	Yes
CF	Committed chunks	Ascending differentiable chunks; retained graph; full backward	Reused	Yes
Async	Native asynchronous pipeline	Native training with rollout across policy versions	Not reused	No
Chunk	Committed chunks	Reverse chunks with prefix adjoints and KV transfer scheduling	Reused	Yes

to-training transition, union Sync puts rollout replicas into level-2 sleep and the teacher into level-1 sleep, then loads the offloaded trainer state. After training, weight publication restores rollout weights and cache availability, and the teacher wakes before the next batch. Dedicated Sync also uses level-2 rollout sleep at the batch boundary, while its teacher remains on a separate GPU pool. These transitions and trainer offload/load costs are included in the reported update time.

Chunked forward. CF retains Streaming Chunk Scoring and rollout KV reuse, and consumes rollout tokens in ascending chunks on its dedicated trainer pool. Both its teacher submission chunk size and student forward chunk size are 1024 tokens, with loss evaluation in blocks of 2048 positions. Its memory planner selects local batches of 8 and 4 for the 4B student at caps 4096 and 8192, respectively; completed 8B runs use 2 and 1. These local batches refer to each trainer rank, rather than the full policy batch. It retains the differentiable prefix KV and activations required for a full backward pass across the microbatch's chunks. Activation checkpointing recomputes intermediates within checkpointed layers, while cross-chunk graph dependencies remain live until backward. CF computes the local loss once its targets are complete and accumulates gradients over the full policy batch before updating weights. CF normalizes microbatch gradients with a common provisional denominator, then rescales them using the final valid-token count. Planning uses the maximum trajectory length because final lengths are unavailable when streamed forward computation begins.

Asynchronous execution. Async overlaps rollout and native student training across policy versions. Table 8 lists the configuration used for the short performance runs and the end-to-end case study. Both use verl's built-in `ReplayBufferAsync`, one warmup prompt batch, one optimizer update and weight publication per consumed batch, and an age threshold of eight with the `drop`

strategy. The threshold applies to terminal prompt-group age at sampling time, which differs from the number of policy versions used to generate a trajectory. The shared TransferQueue tracks pending, running, completed, and failed prompt groups, and there is no separate fixed replay-capacity parameter in this configuration. The sampler polls the queue every two seconds and selects a batch once enough eligible completed groups are available. Custom sampling, DAPO group filtering, and hybrid trainer-to-rollout switching are disabled.

Table 8: Async baseline configuration. Sequence limits are vLLM `max_num_seqs`; batched tokens are `max_num_batched_tokens`.

Setting	Short performance	200-update case study
Trainer / placement	`separate_async`; dedicated allocations in Table 5	`separate_async`; dedicated T2/R1/H1
Batch / synchronization	128 prompt groups, one trajectory each; one optimizer update and weight publication per batch	64 prompt groups, four trajectories each; one optimizer update and weight publication per batch
Queue and sampler	One warmup prompt batch; built-in `ReplayBufferAsync`; 2-second polling; no fixed-capacity replay parameter	Same
Age control	Threshold 8, `drop`; no custom sampler or group filtering	Same
Rollout vLLM	Async mode; TP1; memory fraction 0.5; 256 sequences; 8192 batched tokens	Same
Teacher vLLM	1024 sequences; 8192 batched tokens; memory fraction 0.5, except 0.9 for selected four-GPU 4B/32B runs; TP follows Table 5	TP1; memory fraction 0.5; 1024 sequences; 8192 batched tokens
Student training	Native FSDP; activation checkpointing and dynamic token batching; no parameter or optimizer offload; hybrid trainer-to-rollout switching disabled	Same

During weight publication, the native checkpoint manager can interrupt unfinished requests and release their KV. The client keeps the generated token IDs and resumes from the prompt and retained response prefix, repeating prefill under the newly published weights. Groups that exceed the configured age threshold can be removed and replaced. The saved performance and end-to-end records have no eviction counter, so the number of removals is unknown. Internally interrupted requests can resume without returning an aborted response. The returned-response aborted ratio is zero in every measured short-run update and every end-to-end update.

For a consumed trajectory, version span counts the distinct student versions used during generation, and staleness is the lag from the training version to the freshest generation version. Table 9 reports these observed quantities. For the short experiments, mean lag averages the per-update trajectory means over the 36 measured updates from 12 completed main-matrix runs. For end-to-end training, it averages the corresponding values over all 200 updates.

Table 9: Realized Async policy-version behavior. Update-maximum columns give the number of updates with each maximum value; Aborted is the maximum returned-response aborted ratio.

Experiment	Updates	Mean lag	Update maximum lag	Update maximum span	Aborted
Short performance	36	0.759	1: 36 updates	1: 31; 2: 5 updates	0
End-to-end	200	0.600	0: 1; 1: 192; 2: 7 updates	1: 20; 2: 180 updates	0

The version statistics describe Async's consumed trajectories; update time also includes renewed prefill, queue waiting, and native student training. Async provides a deployment reference; Sync and CF, like ChunkOPD, keep a fixed policy version throughout each batch.

B.4 MEASUREMENT DEFINITIONS AND RESULT RECORDS

For short performance runs, we discard one warmup update and average the next three updates. The tables report these means; the result records retain individual step values and sample dispersion. Numerical tables report means and sample standard deviations across five trajectory groups. End-to-end curves use the complete saved training and checkpoint records described in Appendix F.

Table 10: Definitions used in the performance analysis.

Metric	Definition
OPD update time	Elapsed time of the evaluated distillation update, including rollout, teacher scoring, student optimization, weight publication, KV handling, and scheduler lifecycle work.
Rollout completion	Time from the policy/scheduler timing origin until all rollout trajectories terminate.
Teacher tail	Additional elapsed time after rollout completion until all teacher targets are ready; zero if teacher scoring has already completed.
Teacher busy time	Accumulated teacher service time; concurrent work can overlap the rollout interval.
Student training	Recorded actor-update interval, including the work within that trainer invocation.
Allocated memory	Peak allocated tensor memory recorded by the trainer allocator.
Reserved memory	Peak memory reserved by the allocator, including unused cached blocks.
Response throughput	Total generated response tokens divided by the total reported update time in the same window.

Stage plots show rollout completion, teacher tail, and training; weight publication and other lifecycle costs remain included in complete update times. Teacher service and copies can overlap other work; their durations cannot simply be added to elapsed compute time. Appendix D reports actual response lengths. The research records retain resolved configurations, source identifiers, commands, logs, and the scripts used to generate figures and tables.

C NUMERICAL CONSISTENCY AND GRADIENT DIAGNOSTICS

We compare teacher scores, student KV caches, and parameter gradients across execution paths. We hold trajectories fixed so that changes in generated data do not affect the comparison. We compare saved gradients before clipping or optimization; Appendix F reports the training results.

C.1 FIXED-TRAJECTORY PROTOCOL AND METRICS

The audit uses a Qwen3-0.6B student and Qwen3-4B teacher, eight GPUs, and a total-token cap of 4096. We generate five disjoint groups of 128 trajectories from consecutive blocks of DAPO-Math rows 0 to 639, using greedy decoding and seed one. Together they contain 640 trajectories and 2,537,219 valid response targets. For each path, we reset gradients to zero and use the same student parameters, fixed trajectories, and saved teacher targets. The native reference evaluates the model and loss directly, without the full synchronous scheduler. Reverse paths use chunk sizes 512 and 2048, with either HF forward KV or the exported rollout KV.

For a tensor or concatenated gradient g and reference g_0, we report

$$e_{\mathrm{rel}} = \frac{\|g - g_0\|_2}{\|g_0\|_2}, \qquad c = \frac{\langle g, g_0 \rangle}{\|g\|_2 \|g_0\|_2}. \tag{6}$$

We compute discrepancies within each trajectory group before averaging the statistics equally across the five groups. The tables report sample standard deviations across groups, using denominator four. Since the groups are consecutive data blocks, these standard deviations measure batch-to-batch variation within the dataset, not variation across independent training seeds. Table 11 lists the group sizes and gradient comparisons.

Table 11: Fixed-trajectory groups and relative gradient L2 differences from native BF16 training. The two reverse columns use rollout KV.

Group	Dataset rows	Targets	Repeat (%)	$C = 512$ (%)	$C = 2048$ (%)
0	0–127	508,444	0.207	3.864	4.162
1	128–255	511,800	0.210	6.352	4.123
2	256–383	504,643	0.199	3.808	3.823
3	384–511	508,599	0.206	4.815	5.013
4	512–639	503,733	0.210	4.660	4.438

C.2 TEACHER-SCORE CONSISTENCY

We compare streaming and complete-trajectory teacher scoring on the same retained response contexts. Top-1 agreement is the fraction of prediction rows with the same highest-probability token. Top-k overlap averages the fraction of retained vocabulary entries shared by the two supports. We compute log-probability MAE and RMSE on shared entries. For the mean group-maximum absolute error, we take the largest absolute error in each group and average the five maxima. We repeat complete-trajectory scoring to measure execution variability.

Table 12: Teacher-score agreement on the five fixed-trajectory groups. Values are mean $\pm$ sample standard deviation; probability agreements are percentages.

Metric	Complete repeat	Streaming / complete
Top-1 agreement (%)	99.9950 ± 0.0041	99.5543 ± 0.0299
Top-k overlap (%)	99.9908 ± 0.0080	99.2256 ± 0.0388
Shared log-probability MAE	0.000859 ± 0.000742	0.072866 ± 0.003221
Shared log-probability RMSE	0.012393 ± 0.008050	0.149042 ± 0.007544
Mean group-maximum absolute error	5.1625 ± 4.0048	17.5243 ± 3.1494

The two scoring paths agree closely on retained predictions, although their log-probabilities differ (Table 12). For prediction row r in a trajectory group G, let q_r^{c} and q_r^{s} denote the complete-trajectory and streaming teacher probabilities, with top-k supports S_r^{c} and S_r^{s}. Define $\widetilde{q}_r^{a}(v) = q_r^{a}(v)\mathbf{1}\{v \in$

$S_r^a\}$ for $a \in \{\mathrm{c}, \mathrm{s}\}$, setting unretained entries to zero without renormalization. The group-mean top-k total variation is the mean half-L1 difference between these truncated probability vectors:

$$\mathrm{TV}_k(G) = \frac{1}{2|G|} \sum_{r \in G} \sum_{v \in S_r^{\mathrm{c}} \cup S_r^{\mathrm{s}}} \left| \widetilde{q}_r^{\mathrm{c}}(v) - \widetilde{q}_r^{\mathrm{s}}(v) \right|. \tag{7}$$

For shared entries $I_r = S_r^{\mathrm{c}} \cap S_r^{\mathrm{s}}$, let $w_r(v) = [q_r^{\mathrm{c}}(v) + q_r^{\mathrm{s}}(v)]/2$. The probability-weighted shared-log-probability absolute error normalizes by the total shared-entry weight within each group:

$$E_{\mathrm{w}}(G) = \frac{\sum_{r \in G} \sum_{v \in I_r} w_r(v) \left| \log q_r^{\mathrm{c}}(v) - \log q_r^{\mathrm{s}}(v) \right|}{\sum_{r \in G} \sum_{v \in I_r} w_r(v)}. \tag{8}$$

We average each group-level metric equally across the five groups, obtaining top-k total variation 0.004062 and probability-weighted shared-log-probability absolute error 0.008131. The mean within-group maximum token total variation is 0.223802, so some contexts have larger deviations despite the close aggregate agreement. MAE covers only shared entries and should be read together with support overlap. All comparisons match prediction positions as defined in Appendix A.1 and use finite-precision scoring.

C.3 KV-CACHE CONSISTENCY

We compare keys and values separately across the rollout cache, HF BF16 forward KV, and an FP32 reference. Repeated HF BF16 forward passes produce identical KV in this audit. Table 13 reports the relative L2 differences, including between-group dispersion. Rollout versus HF BF16 errors average 0.785% for keys and 2.438% for values.

Table 13: KV-cache relative L2 differences (%). Values are mean ± sample standard deviation across five groups.

Comparison	K rel. L2 (%)	V rel. L2 (%)
HF BF16 repeat	0.000 ± 0.000	0.000 ± 0.000
Rollout / HF BF16	0.785 ± 0.032	2.438 ± 0.061
HF BF16 / FP32	0.675 ± 0.029	2.054 ± 0.056
Rollout / FP32	0.522 ± 0.028	1.574 ± 0.060

The FP32 path uses SDPA, whereas the BF16 paths use FlashAttention 2. Comparisons to FP32 therefore include both arithmetic precision and attention-backend changes. Their outputs can differ even with identical parameters and token sequences. We next compare parameter gradients from the complete training computation.

C.4 GRADIENT CONSISTENCY ACROSS EXECUTION PATHS

Table 14 compares the concatenated parameter gradients against native BF16 training. Reverse training with HF KV and reverse training with rollout KV both retain causal prefix-gradient propagation. We use the same trajectories and teacher targets for both cache sources. Gradient differences remain with HF KV, so rollout KV alone does not account for the discrepancy.

Table 14: Gradient differences from native BF16 training. Values are five-group mean ± sample standard deviation. Relative L2 uses the norm of the concatenated native parameter gradient.

Training path	Overall L2 (%)	Cosine
Native repeat	0.206 ± 0.005	0.999998 ± 0.000000
HF KV, $C = 512$	3.436 ± 0.551	0.999398 ± 0.000194
Rollout KV, $C = 512$	4.700 ± 1.029	0.998853 ± 0.000524
HF KV, $C = 2048$	3.862 ± 0.515	0.999244 ± 0.000207
Rollout KV, $C = 2048$	4.312 ± 0.448	0.999062 ± 0.000200

The rollout-KV paths have overall relative L2 differences of 4.700% and 4.312% at chunks 512 and 2048, respectively, and five-group mean cosine similarities above 0.9988. The native repeat differs

by 0.206%, so the reverse-path discrepancy exceeds native run-to-run variation. The larger chunk reduces the discrepancy with rollout KV but not with HF KV. Thus, increasing chunk size does not consistently improve gradient agreement across the tested paths.

Table 15 compares the same paths with the FP32/SDPA gradient. Native BF16 itself has a 3.305% relative L2 difference from that reference. Relative errors in this table use the norm of the FP32/SDPA gradient as the denominator; Table 14 instead compares directly with native BF16. As with KV, the FP32 comparison changes both precision and the attention backend.

Table 15: Gradient comparison against the FP32/SDPA reference. Mean $\pm$ sample standard deviation across five groups.

Training path	**Relative L2 (%)**	**Cosine**
Native BF16	3.305 ± 1.188	0.999397 ± 0.000474
HF KV, $C = 512$	4.329 ± 0.760	0.999040 ± 0.000334
Rollout KV, $C = 512$	4.931 ± 1.617	0.998680 ± 0.000830
HF KV, $C = 2048$	4.773 ± 1.268	0.998796 ± 0.000705
Rollout KV, $C = 2048$	4.549 ± 1.070	0.998919 ± 0.000550

C.5 Variation across trajectory groups

Table 11 reports the overall gradient discrepancies for each fixed-trajectory group. For rollout KV at chunk 512, the overall relative L2 ranges from 3.808% to 6.352%; at chunk 2048 it ranges from 3.823% to 5.013%. Both ranges use the concatenated parameter gradients for each group.

The exact-arithmetic derivation establishes that reverse traversal preserves all gradient dependencies when the forward values satisfy the stated conditions. The BF16 implementations still differ numerically, as the fixed-trajectory comparisons show. These measurements precede clipping and optimization; Appendix F separately examines the learning trajectories.

D EXTENDED PERFORMANCE RESULTS

We report performance across model sizes, GPU budgets, trajectory lengths, batch sizes, and teacher architectures using the configurations in Appendix B. All comparisons use the same top-32 objective. We also report actual response lengths, since generation can stop before the token cap.

D.1 DENSE-MODEL LATENCY AND REALIZED LENGTHS

Table 16 reports the full dense-model matrix: four student-teacher pairs, two token caps, and two physical GPU budgets. OOM marks configurations that fail because of insufficient trainer memory. ChunkOPD and Sync complete all 16 settings, and ChunkOPD is 1.11 to 1.79× faster, as reported in the main text. The table also includes the CF and Async runs that complete under the same physical GPU budgets.

Table 16: Complete dense-model OPD update times (seconds), batch 128. OOM denotes a trainer-memory failure that prevents a complete update.

Model pair	Cap	Chunk	Sync	CF	Async
4 physical GPUs; batch 128					
4B/14B	4096	**112.9**	192.8	146.6	169.8
4B/14B	8192	**278.4**	418.3	520.1	422.8
4B/32B	4096	**150.4**	189.2	245.9	158.1
4B/32B	8192	**375.4**	418.6	645.1	432.8
8B/14B	4096	**163.0**	252.8	OOM	OOM
8B/14B	8192	**416.1**	572.4	OOM	OOM
8B/32B	4096	**194.0**	307.2	OOM	OOM
8B/32B	8192	**507.4**	650.8	OOM	OOM
8 physical GPUs; batch 128					
4B/14B	4096	**85.5**	117.0	94.7	109.6
4B/14B	8192	**196.9**	242.8	274.9	211.2
4B/32B	4096	**92.1**	137.2	110.2	108.1
4B/32B	8192	**208.3**	275.0	279.1	208.4
8B/14B	4096	**129.5**	221.7	137.3	184.2
8B/14B	8192	**280.7**	464.3	407.9	399.0
8B/32B	4096	**135.4**	242.7	142.0	204.3
8B/32B	8192	**322.5**	504.0	418.0	416.5

Table 17 gives the mean response lengths for the paired union-placement comparisons. We average lengths over the same three steps used to measure latency. ChunkOPD has lower latency despite slightly longer mean responses in every listed pair. The bottom four rows report lengths for the batch-size extension, which uses the same total-token budgets.

D.2 UNION AND DEDICATED DEPLOYMENTS

Table 18 compares the dense 8B/32B pair across union and dedicated deployments. Both deployments preserve the same full-batch update boundary for Sync and ChunkOPD. Dedicated placement permits training of completed units while other inference work remains active; union placement makes the full GPU set available during student training. Async is evaluated with dedicated placement because its rollout and training overlap spans policy versions, while the table provides same-placement Sync and ChunkOPD controls. All entries use the same model workload.

On eight GPUs, dedicated ChunkOPD takes 137.93 and 411.41 seconds at caps 4096 and 8192, compared with 142.00 and 417.99 seconds for CF under the same T4/R2/H2 allocation. Both methods use rollout KV, but CF retains the graph from ascending forward chunks, whereas Reverse Chunk Training recomputes chunks in reverse order. Thus, the runs compare training schedules and their memory and batching plans with the same Streaming Chunk Scoring, KV reuse, and GPU allocation. With the MoE teacher under the same dedicated allocation, the corresponding CF/ChunkOPD times are 136.7/134.5 and 396.0/370.6 seconds. The training components are the same in these dedicated runs and the main union-placement comparison.

Table 17: Mean response lengths in the union-placement experiments. Difference is ChunkOPD/Sync -1; caps constrain total prompt-plus-response length.

Model pair	GPUs	Cap	Batch	Sync length	Chunk length	Difference
4B/14B	4	4096	128	3045.4	3065.7	+0.67%
4B/14B	4	8192	128	6363.3	6505.4	+2.23%
4B/32B	4	4096	128	3040.6	3064.6	+0.79%
4B/32B	4	8192	128	6344.6	6506.3	+2.55%
8B/14B	4	4096	128	3064.8	3070.4	+0.18%
8B/14B	4	8192	128	6882.2	7012.4	+1.89%
8B/32B	4	4096	128	3067.7	3070.3	+0.09%
8B/32B	4	8192	128	6894.8	7033.1	+2.01%
4B/14B	8	4096	128	3048.9	3061.2	+0.40%
4B/14B	8	8192	128	6356.5	6535.3	+2.81%
4B/32B	8	4096	128	3046.1	3064.8	+0.61%
4B/32B	8	8192	128	6358.2	6499.3	+2.22%
8B/14B	8	4096	128	3068.7	3070.0	+0.04%
8B/14B	8	8192	128	6871.6	7003.6	+1.92%
8B/32B	8	4096	128	3066.6	3070.0	+0.11%
8B/32B	8	8192	128	6927.3	7025.5	+1.42%
8B/32B	4	4096	256	3066.5	3071.5	+0.16%
8B/32B	4	8192	256	6890.3	6996.0	+1.53%
8B/32B	8	4096	256	3065.4	3070.1	+0.15%
8B/32B	8	8192	256	6902.3	6953.8	+0.75%

Table 18: Placement comparison for Qwen3-8B/32B, batch 128. Values are OPD seconds per update; dash denotes an unavailable configuration and OOM denotes teacher prompt-logprob CUDA out-of-memory.

GPUs	Cap	Sync union	Sync dedicated	Chunk union	Chunk dedicated	CF dedicated
4	4096	307.2	359.1	194.0	285.9	—
4	8192	650.8	OOM	507.4	818.6	—
8	4096	242.7	264.7	135.4	137.9	142.0
8	8192	504.0	551.4	322.5	411.4	418.0

D.3 Batch size and throughput

Table 19 reports update time and throughput for batches of 128 and 256 trajectories. For batch B, mean update time $\bar{T}$, and mean response length $\bar{L}$ over the measurement window, the reported rates are $B/\bar{T}$ and $B\bar{L}/\bar{T}$. We compute both rates over the same trajectories and measurement window as the latency entry. Response throughput counts each generated response token once, regardless of its subsequent use in training.

Table 19: Batch-size extension for Qwen3-8B/32B under union placement. Seq/s counts completed trajectories per second; Tok/s counts response tokens per second. Times are seconds per update.

GPUs	Cap	Batch	Sync s	Seq/s	Tok/s	Chunk s	Seq/s	Tok/s
4	4096	128	307.2	0.417	1278	194.0	0.660	2026
4	4096	256	636.9	0.402	1233	355.7	0.720	2210
4	8192	128	650.8	0.197	1356	507.4	0.252	1774
4	8192	256	1383.9	0.185	1275	882.8	0.290	2029
8	4096	128	242.7	0.527	1617	135.4	0.945	2903
8	4096	256	407.3	0.629	1927	206.0	1.243	3816
8	8192	128	504.0	0.254	1759	322.5	0.397	2789
8	8192	256	826.0	0.310	2139	511.7	0.500	3479

Doubling the batch increases ChunkOPD's update time by less than twofold in every configuration, raising trajectory throughput. The larger batch also raises its speedup over Sync in all four paired

settings, reaching 1.98× on eight GPUs at cap 4096. Token throughput also depends on actual response length, especially at cap 8192, where many responses terminate before reaching the cap.

D.4 MoE teacher results

The MoE extension replaces the dense teacher with Qwen3-30B-A3B while retaining the Qwen3-8B student and batch 128. Table 20 reports the union-placement results on both GPU budgets. ChunkOPD is 1.55 to 1.86× faster in the tested configurations. Both CF and ChunkOPD also complete the dedicated runs with this teacher, as reported above.

Table 20: MoE teacher extension: Qwen3-8B student and Qwen3-30B-A3B teacher, batch 128, union placement. Times are seconds per update.

GPUs	Cap	Sync	Chunk	Speedup
4	4096	280.4	181.4	1.55×
4	8192	744.9	422.8	1.76×
8	4096	244.8	131.3	1.86×
8	8192	506.7	282.0	1.80×

D.5 Execution feasibility and failure stages

All eight failures in the main matrix occur in four-GPU CF or Async runs with an 8B student. They result from insufficient trainer memory, so we report no latency for those runs. The separate four-GPU dedicated 8B/32B control at cap 8192 has a teacher prompt-logprob CUDA OOM for Sync, while ChunkOPD completes.

These failures apply to the listed configurations and resource allocations.

E COMPONENT CONTRIBUTIONS AND PERFORMANCE ANALYSIS

We measure the latency contribution of each component and examine how rollout duration, teacher scoring, chunk size, and local batch affect performance. We also report cache volumes and transfer counters to quantify KV movement within an update.

E.1 COMPONENT-REMOVAL ABLATIONS

The ablation *w/o Streaming Chunk Scoring* submits one complete teacher prefill after each trajectory terminates. Keeping the rest of the pipeline unchanged isolates the effect of scoring fragments before the trajectory finishes. The ablation *w/o Reverse Chunk Training* uses the native training worker and disables rollout KV export and reverse slots together. This removes the entire training component, including rollout KV reuse and the associated transfer schedule. The ablation includes the operator and memory-management differences described in Appendix B.

Tables 21 and 22 report full-update times under union and dedicated placement, respectively. Removing Streaming Chunk Scoring increases measured step time by 9.9 to 28.1% across the eight settings under union placement.

Table 21: Union-placement component ablations, cap 4096 and batch 128. Relative time is variant/ChunkOPD.

		4 GPUs		8 GPUs	
Model pair	**Variant**	**Time (s)**	**Relative**	**Time (s)**	**Relative**
4B/14B	Chunk	**112.9**	**1.00×**	**85.5**	**1.00×**
	w/o Streaming Chunk Scoring	140.9	1.25×	95.8	1.12×
	w/o Reverse Chunk Training	125.4	1.11×	98.2	1.15×
4B/32B	Chunk	**150.4**	**1.00×**	**92.1**	**1.00×**
	w/o Streaming Chunk Scoring	176.0	1.17×	117.9	1.28×
	w/o Reverse Chunk Training	156.3	1.04×	105.5	1.15×
8B/14B	Chunk	**163.0**	**1.00×**	**129.5**	**1.00×**
	w/o Streaming Chunk Scoring	187.8	1.15×	142.3	1.10×
	w/o Reverse Chunk Training	180.6	1.11×	203.3	1.57×
8B/32B	Chunk	**194.0**	**1.00×**	**135.4**	**1.00×**
	w/o Streaming Chunk Scoring	225.2	1.16×	162.4	1.20×
	w/o Reverse Chunk Training	248.5	1.28×	210.3	1.55×

Table 22: Measured dedicated-placement component ablations, eight GPUs, T4/R2/H2, cap 4096, and batch 128. Values are seconds per update.

Pair	**Chunk**	**w/o Streaming Chunk Scoring**	**w/o Reverse Chunk Training**
4B/14B	**96.36**	101.42	101.30
4B/32B	**112.83**	145.70	135.48
8B/14B	134.12	**133.25**	135.17
8B/32B	**137.93**	167.79	170.52

Under dedicated placement, the benefit of each component depends on the workload. For 8B/32B, removing Reverse Chunk Training raises measured latency by 23.6%. For 8B/14B, the differences are small, and *w/o Streaming Chunk Scoring* is slightly faster than the complete pipeline. Reducing one stage's service time may leave update time unchanged if another stage finishes last. The stage measurements below help explain this dependence on workload.

E.2 STAGE-TIME BREAKDOWN AND SEQUENCE-LENGTH BALANCE

Table 23 extends the main-text stage plot to both token caps. The rollout interval R ends when all trajectories terminate. The teacher tail H measures the additional elapsed time after rollout comple-

tion until all teacher targets are ready, and is zero if teacher scoring has already completed. Teacher computation that overlaps rollout falls within R, so $R + H$ gives the time to inference completion from the same timing origin. The training column uses the actor-update timer. The complete update time also includes weight publication and lifecycle work outside these three intervals.

Table 23: Stage times for all dense union-placement configurations, batch 128. R: time to rollout completion; H: additional teacher-scoring time after rollout completion; T: student training. All values are seconds; update time also includes weight publication and other lifecycle costs.

GPUs	Pair	Cap	Sync R	Sync H	Sync T	Chunk R	Chunk H	Chunk T
4	4B/14B	4096	57.3	84.0	36.8	57.4	9.4	29.2
4	4B/14B	8192	235.7	88.5	79.4	200.0	6.5	54.4
4	4B/32B	4096	57.6	78.7	36.9	58.0	42.8	30.9
4	4B/32B	8192	238.5	81.5	80.4	200.6	100.1	56.5
4	8B/14B	4096	70.8	85.3	70.4	71.2	8.7	50.6
4	8B/14B	8192	289.4	112.8	146.1	271.9	8.0	102.3
4	8B/32B	4096	70.6	89.4	102.3	71.0	42.9	49.9
4	8B/32B	8192	289.6	114.9	206.7	272.6	105.0	98.4
8	4B/14B	4096	41.7	21.2	31.0	44.0	1.1	18.8
8	4B/14B	8192	132.7	25.2	61.9	142.6	0.4	34.0
8	4B/32B	4096	41.8	41.0	31.1	44.8	7.6	18.4
8	4B/32B	8192	131.3	60.1	60.5	137.6	17.5	33.4
8	8B/14B	4096	55.4	21.9	108.4	58.2	0.5	34.8
8	8B/14B	8192	199.6	11.0	218.5	185.4	0.5	59.4
8	8B/32B	4096	55.7	41.1	110.0	57.5	7.7	35.1
8	8B/32B	8192	201.3	50.8	215.2	184.8	40.5	59.2

Longer trajectories. For Qwen3-4B/32B on four GPUs, increasing the total-token cap from 4096 to 8192 changes the Sync-to-ChunkOPD speedup from 1.26× to 1.11×. The absolute update-time reduction still grows from 38.9 to 43.1 seconds. Student-training savings grow from 6.0 to 23.8 seconds, but inference savings, $(R + H)_{\mathrm{Sync}} - (R + H)_{\mathrm{Chunk}}$, fall from 35.5 to 19.3 seconds. The planned safe teacher KV capacity remains similar at the two caps (293,696 and 290,432 tokens), but the larger reservation per trajectory reduces the planned active-trajectory limit from 71 to 35. Both limits are below the teacher engine's sequence ceiling of 128, so the KV-capacity plan controls admission. ChunkOPD's teacher tail grows from 42.8 to 100.1 seconds. At cap 8192, rollout completes at 200.6 seconds for ChunkOPD and 238.5 seconds for Sync; their respective teacher tails are 100.1 and 81.5 seconds. Inference therefore finishes at 300.7 and 320.0 seconds, a difference of 19.3 seconds. Although longer rollout provides more time for overlap, the teacher must score longer contexts with fewer trajectories admitted at once. The net inference saving depends on both effects.

E.3 Chunk-size sensitivity

Figure 8 summarizes the teacher and reverse-training sweeps on Qwen3-4B/14B with four GPUs, batch 128, and cap 8192. We manually specify chunk sizes for these sweeps. Outside these sweeps, the main matrix uses a fixed teacher chunk size of 1024, while the reverse planner selects the training chunk size and local batch from available memory.

E.3.1 Teacher chunk size

Table 24 reports teacher busy time, teacher tail, and total inference time for the teacher chunk-size sweep. The chunk-512 and chunk-2048 measurements execute rollout, teacher scoring, KV export, and the union-placement inference lifecycle without trainer compute. The chunk-1024 measurement uses the inference interval of the complete pipeline. All three measurements stop at inference completion and exclude trainer compute.

Larger chunks require fewer submissions and spread execution overhead over more tokens. Teacher busy time decreases from 116.60 to 96.49 seconds as the chunk grows from 512 to 2048. Inference makespan stays between 206.52 and 209.06 seconds, while the remaining tail stays below 7.5 seconds. Generation occupies most of the critical path on this workload, so lower teacher busy time has little effect on inference completion time.

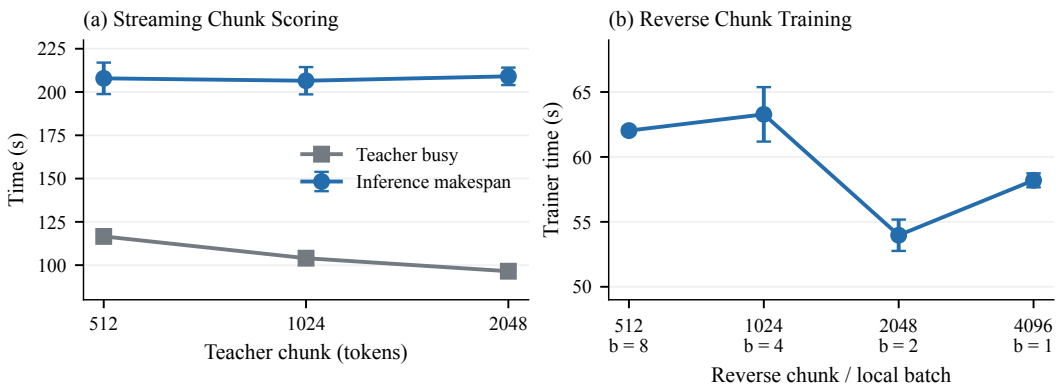


Figure 8: **Chunk-size sensitivity.** Qwen3-4B/14B, four GPUs, batch 128, cap 8192. (a) Teacher scoring; (b) reverse training with co-tuned local batch b. Points are means; Section E.3.2 reports memory measurements and fixed-batch results.

Table 24: Teacher chunk-size sensitivity on the 4B/14B workload. C_H is the scoring chunk size; all timing columns are seconds.

C_H	**Teacher busy (s)**	**Teacher tail (s)**	**Inference (s)**
512	116.60	6.35	207.89
1024	104.01	6.48	206.52
2048	96.49	7.44	209.06

E.3.2 Reverse chunk size, local batch, and memory

The reverse sweep runs the complete pipeline and measures the actor-update interval. For each manually specified chunk size, the planner chooses the largest feasible power-of-two local batch under a common batch cap of 128. The four plans all have $C_R b = 4096$ tokens: chunk sizes 512, 1024, 2048, and 4096 pair with local batches 8, 4, 2, and 1. Figure 8(b) plots their training times, and Table 25 gives peak allocator memory.

Table 25: Reverse chunk-size sensitivity with co-tuned local batch. Times are mean seconds per actor update; memory is the recorded peak allocated/reserved footprint in GiB.

C_R	b	**Trainer (s)**	**Allocated GiB**	**Reserved GiB**
512	8	62.03	59.51	74.98
1024	4	63.28	50.94	63.67
2048	2	53.97	47.59	62.24
4096	1	58.20	48.72	76.36

Figure 9 includes the fixed-local-batch diagnostic at $b = 2$. With the local batch fixed, chunks 512, 1024, and 2048 take 69.48, 62.72, and 54.37 seconds, respectively. At chunk 512, allowing a larger local batch reduces scheduling overhead and brings training time to 62.03 seconds. Among the four primary plans, chunk 2048 with local batch two is fastest at 53.97 seconds. Increasing the chunk to 4096 forces the local batch down to one; peak reserved memory reaches 76.36 GiB and update time rises to 58.20 seconds.

The memory footprint includes parameters, optimizer state, gradient storage, KV backing and adjoints, active-chunk activations, and the vocabulary loss workspace. Chunking limits activation storage to the active chunk, but the causal KV context and its adjoints still span the sequence. The reported peaks include all these terms. The fastest tested plan uses chunk 2048, not the smallest chunk, because performance depends on how chunk size and local batch fit within the memory budget together.

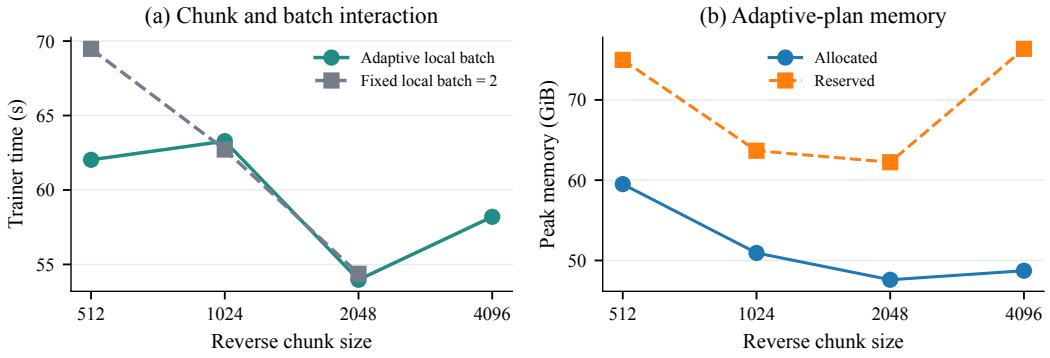


Figure 9: Training time and memory at different chunk sizes. (a) Adaptive-batch and fixed-batch measurements on the same model workload. (b) Peak allocated and reserved memory for the adaptive plans.

E.4 KV HANDLING AND TRANSFER DIAGNOSTICS

Table 26 reports KV export and trainer-slot measurements at cap 4096. Export volume is the rollout-side accumulated cache volume for the update. Slot backing and copy/wait counters are the trainer's recorded metrics, averaged over the three measured steps. The counters cover different operations that can overlap, so their sum would not give critical-path time.

Table 26: KV-handling diagnostics for ChunkOPD, union placement, batch 128, cap 4096. Copy is the recorded slot CUDA-copy duration; the two wait columns are initial and next-slot waits. Volumes and slot backing are GiB; durations are seconds.

GPUs	Pair	Export GiB	Slot GiB	Copy s	Initial wait s	Next wait s
4	4B/14B	55.61	2.25	1.25	0.17	2.87
4	4B/32B	55.59	2.25	1.21	0.17	2.82
4	8B/14B	55.69	2.25	1.14	0.20	2.31
4	8B/32B	55.69	2.25	1.15	0.19	2.34
8	4B/14B	55.53	3.38	0.81	0.24	0.00
8	4B/32B	55.59	3.38	0.78	0.22	0.00
8	8B/14B	55.68	2.25	0.61	0.24	1.07
8	8B/32B	55.68	2.25	0.60	0.19	1.08

At cap 4096, ChunkOPD's rollout intervals are 1.00 to 1.07 times those of Sync, including student KV export. Export can proceed as individual trajectories finish while other trajectories and teacher sessions are still active. Within the trainer, suffix pages for the next microbatch can load while backward continues on the current prefix (Appendix A.4). Initial and next-slot wait counters record delays that remain after this overlap. A zero wait counter can still accompany nonzero transfer cost because other loading and synchronization intervals have separate counters.

In the chunk sweep, changing from chunk 2048/local batch two to chunk 4096/local batch one doubles the per-rank microbatch count and increases recorded KV prefetch transfer time from 3.16 to 6.60 seconds. Because these copies can overlap computation, the increase in copy time need not translate into the same increase in training time. Cache volumes measure how much data moves, while the copy and wait counters distinguish transfer activity from exposed delays.

F END-TO-END TRAINING AND EVALUATION

We train Sync, ChunkOPD, and Async for 200 updates and compare their learning trajectories and training time. We report all evaluated checkpoints and training curves, then recompute final scores after removing known overlaps between training and evaluation data. All plots and statistics come from saved training logs and generated responses.

F.1 TRAINING AND EVALUATION SETUP

The training recipe is adapted from Li et al. (2026b). All three methods start from Qwen3-1.7B-Base and use a frozen Qwen3-4B-Base-GRPO teacher. The training manifest contains 12,800 unique DAPO-Math-17K-Processed prompts selected after training-length filtering. We checked these prompts for overlap with the evaluation sets; Section F.5 reports the matches and filtered evaluation scores. Each update uses 64 prompts with four responses each, for 256 trajectories. We submit each prompt as four independent requests with rollout $n = 1$ and disable data shuffling. Async can prefetch, reorder, or discard trajectories, so its optimizer batches can differ from those of the strict schedules even with the same source prompt order. It uses the Async queue, sampling, and weight-publication settings in Table 8. Across 200 updates, its mean consumed-trajectory lag is 0.600 policy versions; the update-level maximum is two in seven updates, and trajectories span at most two versions (Table 9).

Table 27: Training and evaluation settings for the end-to-end case study.

Setting	Value
Student / teacher	Qwen3-1.7B-Base / Qwen3-4B-Base-GRPO
Training prompts	DAPO-Math-17K
Training updates / seed	200 / 1
Prompts / rollouts per prompt	64 / 4 per update
Total-token budget	8192 tokens
Training temperature / top-p	1.0 / 1.0
Teacher objective	Top-32 forward KL, full-vocabulary normalization
Optimizer / learning rate	AdamW / 10^{-6}
Compute and KV precision	BF16; FP32 master parameters and optimizer states
Framework	verl + vLLM
Hardware per method	Four A100 80GB GPUs
Sync / Chunk placement	Union, T4/R2/H2, teacher TP2
Async placement	Dedicated, T2/R1/H1, teacher TP1
Evaluation benchmarks	AIME24 / AIME25 / AMC23
Evaluation samples per question	16
Evaluation temperature / top-p	0.7 / 0.95
Metric	avg@16
Evaluation checkpoints	0, 25, 50, 100, 150, 200

Evaluation generation. We evaluate each checkpoint on AIME24, AIME25, and AMC23, which contain 30, 30, and 40 questions, respectively. We use the student tokenizer and saved chat template, placing the raw problem in the user prompt without an additional answer-format instruction. The template does not explicitly disable thinking. We evaluate the teacher with the same generation settings and report its score as a fixed reference. We grade the saved outputs with the rule-based Rethinking-OPD evaluator at commit `ac26e38d`, leaving the optional model verifier disabled. The grader extracts the last boxed answer. The evaluator comes from the reference repository; generation follows the settings listed here.

Quality metrics. For benchmark b with Q_b questions and correctness indicators $c_{q,r}$, we compute

$$\text{avg@16}_b = \frac{1}{Q_b}\sum_{q=1}^{Q_b}\frac{1}{16}\sum_{r=1}^{16} c_{q,r}. \tag{9}$$

AIME mean averages the AIME24 and AIME25 scores equally. Macro gives all three benchmarks equal weight, regardless of question count. Both metrics average sample correctness, whereas

pass@16 counts a question as solved if any sample is correct. We verify that each checkpoint has exactly one record for every benchmark/question/sample tuple. All 19 checkpoint/reference groups contain the expected 1,600 responses, totaling 30,400 saved responses.

F.2 Complete checkpoint results

Table 28 reports every evaluated checkpoint, including the initial student and fixed teacher reference. The scores use the same checked correctness labels as Table 3 in the main text. Step-zero scores vary because evaluation samples different outputs from the same initial student weights. Each learning curve follows one training seed using that method's own generated trajectories.

Table 28: Complete checkpoint results, avg@16 (%). Bold marks the best student for each checkpoint and metric. AIME mean averages AIME24/25; macro averages all three benchmarks.

Step	Method	AIME24	AIME25	AIME mean	AMC23	Macro
0	Sync	2.92	1.04	1.98	24.06	9.34
	Chunk	3.12	1.04	2.08	**25.00**	**9.72**
	Async	**3.96**	**1.67**	**2.81**	21.72	9.11
25	Sync	**5.83**	**4.38**	**5.10**	**37.34**	**15.85**
	Chunk	4.79	3.33	4.06	36.56	14.90
	Async	5.62	2.92	4.27	35.94	14.83
50	Sync	11.88	6.67	9.27	**43.91**	20.82
	Chunk	9.79	7.71	8.75	42.97	20.16
	Async	**12.08**	**9.58**	**10.83**	42.19	**21.28**
100	Sync	11.25	**8.54**	**9.90**	**44.84**	**21.55**
	Chunk	**11.88**	7.92	**9.90**	43.91	21.23
	Async	9.79	7.92	8.85	43.75	20.49
150	Sync	11.88	**8.33**	10.10	**43.91**	21.37
	Chunk	11.25	7.50	9.38	43.59	20.78
	Async	**12.50**	**8.33**	**10.42**	**43.91**	**21.58**
200	Sync	12.71	**9.58**	**11.15**	**45.16**	**22.48**
	Chunk	**12.92**	8.96	10.94	44.53	22.14
	Async	10.21	7.08	8.65	42.66	19.98
—	Teacher (reference)	24.17	20.83	22.50	63.28	36.09

All three methods score higher after training than in their initial evaluations. At update 200, macro avg@16 is 22.48% for Sync, 22.14% for ChunkOPD, and 19.98% for Async. The corresponding AIME means are 11.15%, 10.94%, and 8.65%. Scores fluctuate between checkpoints, with some earlier checkpoints outperforming later ones, so the table includes all evaluations alongside the final result.

F.3 Quality versus measured update time

Figure 10 plots checkpoint scores against cumulative measured OPD update time. Each point shows a checkpoint's score and the measured update time needed to reach it. Lines connect observations and do not represent intermediate evaluations. Cumulative time includes all 200 updates, including the first two, but excludes initialization, checkpoint intervals, and evaluation. It is the sum of OPD update times defined in Appendix B.4, not the full process runtime.

The 200 measured OPD updates take 9.41 hours for ChunkOPD, 15.99 hours for Sync, and 12.97 hours for Async. Mean update times over steps 3 to 200 are 170.08, 289.50, and 235.06 seconds, respectively; the totals also include the first two updates, whose times are not represented by these means. These curves measure the quality and cost of each method's own training run. Generated tokens and training difficulty can differ across methods at the same step.

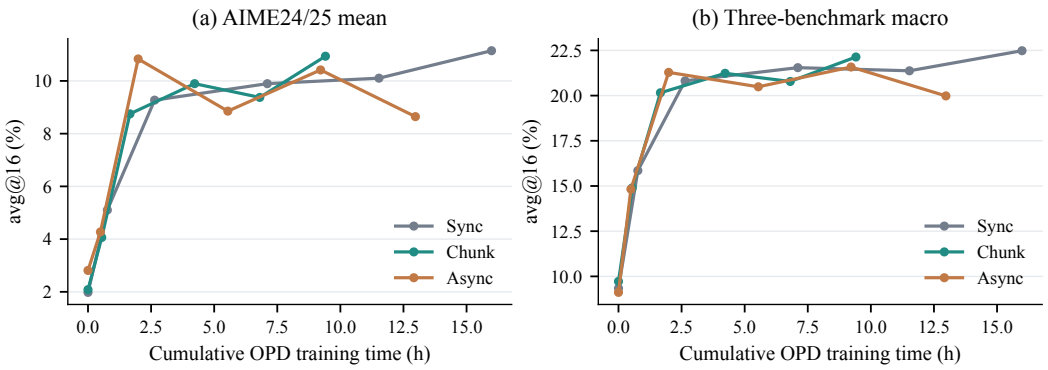


Figure 10: Quality against cumulative measured OPD update time using the saved checkpoint evaluations. (a) AIME24/25 mean. (b) Three-benchmark macro. Each method has one training seed; lines connect the evaluated checkpoints.

F.4 Training dynamics and response lengths

Figure 11 plots training loss, response length, update time, and the fraction of evaluation responses reaching the configured generation length limit. For the per-update metrics, faint curves show all observations and solid curves show trailing five-update means. Each method computes loss on its own generated batches, so matching values do not establish agreement on a fixed dataset. Over the final ten updates, mean top-32 forward-KL losses are 0.1011 for Sync, 0.1011 for ChunkOPD, and 0.1021 for Async.

Training responses become longer over the run. Mean response lengths over the first ten updates are approximately 1,060, 1,031, and 1,016 tokens for Sync, ChunkOPD, and Async; over the final ten they are approximately 4,129, 4,044, and 4,070 tokens. The update times and cumulative costs include this growth in response length. The logs show the length trend but do not identify its cause.

Table 29 reports mean evaluation response lengths and the fraction reaching the configured generation length limit. At update 200, 371/1600 ChunkOPD responses, 379/1600 Sync responses, and 384/1600 Async responses reach this limit. The grader judges a capped response by its extracted answer, so a truncated response can still receive a correct label. All methods use the same evaluation generation budget.

Table 29: Evaluation response lengths and length-cap rates. Mean lengths and cap percentages use all 1,600 responses for each method and checkpoint.

Step	Sync tokens	Cap %	Chunk tokens	Cap %	Async tokens	Cap %
0	1983.6	18.25	1919.8	17.25	1896.5	17.12
25	2391.3	20.12	2335.2	20.06	2380.4	20.56
50	4231.8	22.31	4223.8	21.31	4258.1	21.94
100	4304.1	25.37	4347.8	27.12	4316.3	24.12
150	4273.8	24.88	4297.3	24.69	4244.7	23.12
200	4230.6	23.69	4231.9	23.19	4307.2	24.00

F.5 Overlap between training and evaluation data

The training manifest contains five exact-text matches with AMC23 questions: IDs 3, 15, 19, 30, and 32. The full AMC23 and macro results retain all benchmark questions for consistency with the main-text tables. For Table 30, we remove those five questions from the saved step-200 evaluations and recompute the AMC23 and macro scores. The filtered AMC23 set contains 35 questions and 560 responses per method; AIME24 and AIME25 are unchanged. As in the full-set evaluation, the filtered macro gives all three benchmarks equal weight.

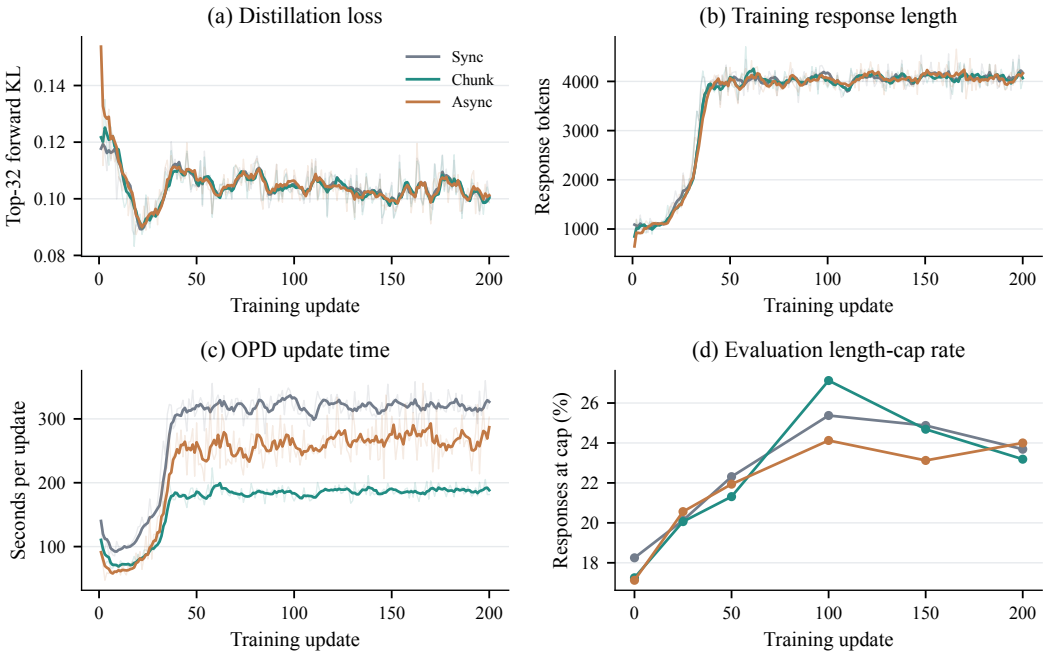


Figure 11: Training and evaluation dynamics over 200 updates. Panels (a) to (c) show per-update values and trailing five-update means. Panel (d) reports truncation of the saved evaluation responses at each checkpoint; training truncation is excluded.

Table 30: Step-200 evaluation after removing the five known training-overlap questions from AMC23, avg@16 (%). Parentheses give the AMC23 question count. All scores use the saved responses, including the fixed teacher reference.

Method	**AMC23 (40)**	**AMC23 (35)**	**Macro (full)**	**Macro (filtered)**
Sync	45.16	44.64	22.48	22.31
Chunk	44.53	44.29	22.14	22.05
Async	42.66	41.43	19.98	19.57
Teacher	63.28	63.57	36.09	36.19

After filtering, AMC23 scores are 44.64% for Sync, 44.29% for ChunkOPD, and 41.43% for Async. The corresponding macros are 22.31%, 22.05%, and 19.57%; the ordering is unchanged. This check covers the known exact-text overlaps but cannot rule out paraphrases or exposure during pretraining. The quality and training-time comparisons remain limited to one training seed per method and the stated evaluation budget.