Rollplex: Cross-Phase GPU Spatial Sharing for Vision Language Model Post-Training

Hanfeng Lu*
HKUST
Hong Kong SAR

Tianyu Feng*
HKUST
Hong Kong SAR

Suyi Li
HKUST
Hong Kong SAR

Yuheng Zhao
HKUST
Hong Kong SAR

Wei Gao
HKUST
Hong Kong SAR

Shaopan Xiong
Alibaba Inc
China

Ju Huang
Alibaba Inc
China

Siran Yang
Alibaba Inc
China

Jiamang Wang
Alibaba Inc
China

Lin Qu
Alibaba Inc
China

Wei Wang
HKUST
Hong Kong SAR

Abstract

Vision-language models (VLMs) enable embodied agents to reason and act from visual observations and language instructions. Reinforcement learning (RL) post-training enhances these capabilities using task feedback, but current on-policy RL runtimes execute rollout, reference scoring, and actor training in strict serial phases. While effective for text-only RL, this phase-granular execution is wasteful for VLMs, where processing dense video inputs and prompt prefixes occupies a large fraction of each phase. Because prefix processing is independent of the generated response, it can be run alongside rollout decoding, which leaves GPU compute capacity underutilized, without breaking synchronous on-policy semantics.

We present Rollplex, a runtime that decomposes the reference and training phase and moves the prefix computation into the rollout decode window. Realizing this schedule requires more than concurrent kernel launches: naive colocation of Qwen2.5-VL-32 B requires roughly 165 GiB per GPU, while rollout and training prefer different tensor-parallel (TP) degrees and weight layouts. Rollplex addresses these constraints with two mechanisms. Phase-aware memory management controls HBM residency according to producer–consumer lifetimes. Parallelism-aware weight sharing uses the same physical storage for layout-compatible tensors across distinct TP degrees and reconstructs only incompatible tensors, avoiding a complete second actor copy. On 32 H800 GPUs, Rollplex achieves 1.23×–1.30× speedup over serial colocation and 1.57×–2.24× over disaggregation under the same GPU budget, while preserving the synchronous RL update.

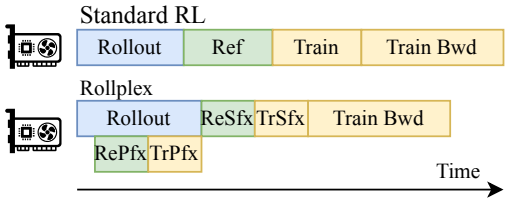

Figure 1. Serial execution and Rollplex's cross-phase spatial schedule. RePfx/ReSfx denote the reference-model prefix/-suffix; TrPfx/TrSfx denote the training-actor prefix/suffix.

1 Introduction

Vision-language models (VLMs) connect perception with language-level reasoning, making them central to embodied artificial intelligence (AI) and robotics. For example, PaLM-E grounds language in sensor observations, while RT-2 and OpenVLA extend VLMs into policies that map observations and instructions to robot actions [3, 6, 17]. Efficient post-training on long visual trajectories is therefore increasingly important.

Current on-policy reinforcement learning (RL) systems for large language models (LLMs) treat rollout, reference scoring, and actor training as atomic scheduling phases. Each iteration generates responses with an actor snapshot θ_k (*rollout*), scores the completed sequences with a frozen reference model, and executes an actor forward, backward, and optimizer step before publishing θ_{k+1}. Production systems [16, 33, 35, 39, 42] distribute these phases across GPUs, but serialize them to preserve the synchronous on-policy

* These authors contributed equally to this work. The order is determined randomly.

dependence. This phase-level schedule is natural when response generation dominates the iteration. However, it becomes unnecessarily coarse for input-heavy VLM RL.

VLM RL changes the resource and time profile of this pipeline. A VLM first encodes video frames into visual tokens and then processes those tokens together with the text prompt through an LLM backbone [2, 9, 50]. Across the four video workloads in this paper, prompt tokens account for 79%–98% of each sampled sequence at the median, compared with 11%–19% for the text-reasoning workloads MATH and GSM8K (§2.2). Prompt-side computation is therefore no longer a small tail: it rivals the autoregressive decode that dominates text-only RL. Although rollout, reference scoring, and actor training perform different computations, each contains substantial prompt-side work. The reference and actor-training prefixes are required by the RL objective, but neither depends on the response that rollout is still generating. Conventional phase ordering nevertheless delays both prefixes until rollout completes.

These prefixes are ready before rollout finishes, but phase-level scheduling prevents them from running. This mismatch exposes an opportunity to accelerate the RL iteration: overlap the rollout-independent prefixes with rollout decode, as Figure 1 illustrates (§2.3). The reference and training-actor prefixes execute while rollout generates the response, and their response-dependent suffixes continue after generation completes. Because this overlap changes only the execution order, it preserves synchronous on-policy semantics without speculating on the response or using a stale policy.

However, existing RL runtimes do not expose this scheduling granularity. Colocated systems execute all phases sequentially on a shared GPU pool [35, 39]. Disaggregated systems place phases on separate GPU pools but preserve the same sequential execution order [16, 41, 53]. Asynchronous and tail-oriented systems increase rollout throughput by relaxing ordering or selecting which responses enter an update [10, 11, 49, 55]. These techniques improve placement, synchronization, or the decode long tail, but none executes response-independent work from a later phase in the spatial slack of the same synchronous rollout.

Naively launching these phases concurrently creates two feasibility challenges and one performance constraint. First, overlap extends tensor lifetimes: rollout KV state, prefix KV caches, training activations, parameters, gradients, and optimizer state together require roughly 165 GiB per GPU for Qwen2.5-VL-32 B, more than twice an H800's 80 GB capacity (§2.4.1). Dropping preserved prefix state saves memory but recomputes the work that overlap tries to hide. Second, rollout and training prefer different tensor-parallel (TP) degrees (§2.4.2). Training uses a wider group to distribute model and optimizer state [31, 36], whereas rollout prefers a narrower group to reduce per-token collectives [18, 28]. A common degree either exceeds training memory or slows decode; separate actor instances duplicate weights and require layout conversion after every update. Third, prefix kernels can contend with decode for SM capacity, cache, and HBM bandwidth. Because later computation waits for the generated response, any decode slowdown extends the critical path and can erase the benefit of overlap. The scheduler must therefore control interference rather than maximize aggregate utilization (§3.1).

We present Rollplex, a runtime that makes phase-level RL scheduling finer grained by decomposing later phases at the first response position (§3). It executes their prefixes during rollout decode, treats decode as the critical path, and preserves their boundary state for the response-dependent suffixes. Two mechanisms enable this schedule:

Phase-aware memory management. Rollplex classifies intermediate state by producer, consumers, and last use. It retains latency-critical boundary KV state, offloads or recomputes bulky training state, releases phase-local rollout and scoring buffers at their last use, and streams the 32-bit floating-point (FP32) optimizer update in chunks. A CUDA-VMM-backed allocator preserves virtual addresses while changing physical residency, allowing the working set to move through HBM without rebuilding engine-level tensor objects.

Parallelism-aware weight sharing. Rollplex backs training and rollout with the same physical actor storage where their layouts can be represented as aliases. It classifies tensors into identical-sharding, transpose-compatible, and sharing-incompatible cases. The first two cases share storage directly or through a transposed logical view; only incompatible tensors are copied and permuted after an update. Thus the engines may use distinct TP degrees without materializing a complete second actor copy.

We implement Rollplex in ROLL [39], with Megatron-Core for training and vLLM for rollout, and evaluate Qwen2.5-VL-32 B on 32 H800 GPUs. Across four video-reasoning workloads, Rollplex achieves 1.23×–1.30× speedup over serial colocation and 1.57×–2.24× over disaggregation under the same GPU budget (§4.2).

This paper makes the following contributions:

1. We characterize VLM RL as an input-heavy regime in which phase-level serialization hides useful parallelism, and formulate the prefix dependence and decode-window bound of a semantics-preserving cross-phase schedule.
2. We design phase-aware memory management that retains critical boundaries in HBM and moves state by producer–consumer lifetime. We also develop parallelism-aware weight sharing that classifies TP layouts, shares compatible storage, and reconstructs only incompatible tensors.
3. We implement Rollplex in ROLL and demonstrate end-to-end gains over serial colocation and disaggregation without relaxing the synchronous on-policy update.

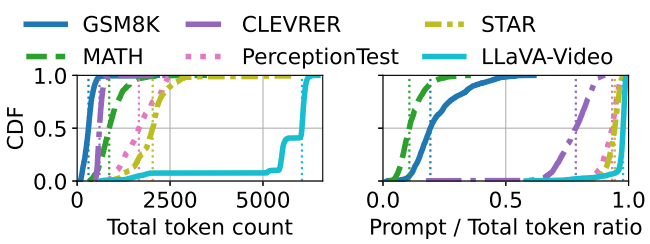

Figure 2. Token statistics for four Video-R1 tasks and the text-RL baselines GSM8K and MATH. **Left**: CDF of total tokens per sample. **Right**: CDF of the prompt fraction.

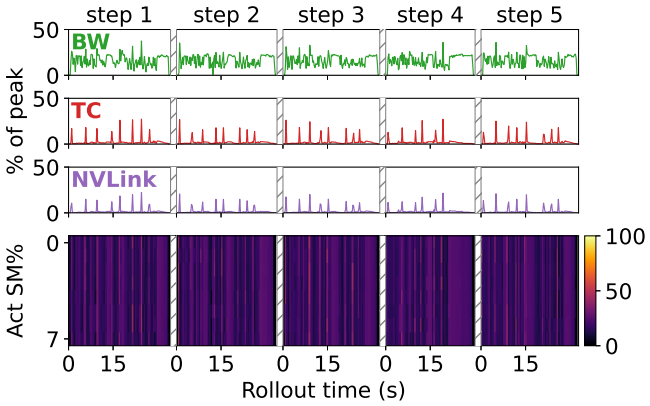

Figure 3. Per-GPU utilization during rollout. TC: Tensor Core; BW: HBM bandwidth.

2 Motivation and Challenges

2.1 Synchronous VLM RL

One synchronous on-policy iteration uses an actor snapshot θ_k in four ordered steps (Figure 1). The rollout engine first samples responses. A frozen reference model then scores the completed sequences to provide the KL regularizer. The training actor computes the RL loss and runs forward and backward passes. Finally, the optimizer updates the actor weights for the next iteration. DeepSpeed-Chat, OpenRLHF, verl, ROLL, and NeMo-Aligner differ in placement and communication, but follow this dependence structure [16, 33, 35, 39, 42]. VLM RL adds a vision transformer (ViT) before the language model. The ViT converts video frames into visual tokens, which the language model processes together with the text prompt. This visual path changes the time profile of each RL phase.

2.2 An Input-Heavy Workload

Text-only RL is usually decode dominated: reference and reward inference together account for less than 15% of step time [11, 41, 55]. The video-intensive VLM workloads change this balance for two reasons. First, visual prompts are long. Even after dynamic-resolution processing, a Qwen-VL encoder at 720p emits roughly 1,536 visual tokens per 16-frame block [2]. Second, the vision encoder and the following long-prompt prefill are dense, compute-intensive operations absent from text-only RL.

Figure 2 quantifies the resulting input-heavy profile. Median prompts contain roughly 470–5,900 tokens on the four video tasks, versus 60 and 90 on GSM8K and MATH. Prompt tokens constitute 79%–98% of a video sample at the median, but only 11%–19% of the text baselines. Because each prompt passes through rollout prefill, reference scoring, and actor training, its cost rivals the autoregressive decode that dominates text RL. Consequently, accelerating decode alone leaves a substantial fraction of a VLM RL iteration untouched. This motivates overlapping prompt-side work with, rather than after, rollout decode.

2.3 Observation and Opportunity

We observe that VLM RL draws training data from two sources: visual inputs and prompts supplied by the dataset, and responses generated during rollout. This boundary splits each model invocation at the first response position into a *prefix* and a *suffix*. Processing the prefix runs the vision encoder and prompt prefill, producing boundary KV state and prefix log-probabilities. Rollout generates the suffix autoregressively; the reference model scores it, and the training phase computes gradients for the model update. The reference and training-actor prefixes depend only on the input and the current model weights, not on the response tokens currently being sampled. They can therefore run before rollout completes without changing the response or the synchronous update. The response-dependent suffixes must wait for generation, but can continue from the retained prefix boundary rather than recomputing the prompt. This independence creates an opportunity to overlap prefix computation with rollout decoding.

Prefix computation is well suited to this overlap because its utilization profile complements rollout decode. Token-by-token decoding repeatedly moves weights and KV blocks while performing little arithmetic per byte, and therefore leaves compute capacity unused [5, 18, 52]. In our H800 profile, active-SM utilization remains below 19% (Figure 3). Video encoding and prompt prefill, in contrast, expose dense batched computation. This complementarity suggests the cross-phase schedule in Figure 1: execute the reference and training-actor prefixes alongside rollout decode, retain their boundary state, and run only the response-dependent suffixes after generation.

The schedule preserves strict on-policy ordering because every overlapped prefix uses the same model weights and input as the serial schedule. Its benefit is capped by the decode window: at most $\min(T_{\text{decode}}, T_{\text{prefix}})$ of prefix work can be removed from the serial critical path. This bound provides

Table 1. Per-GPU footprint of naive overlap for Qwen2.5-VL-32 B.

Role	Component	GiB
Training actor (TP=8)	bfloat16 (BF16) weights	7.6
	FP32 gradient buffer	15.1
	FP32 master weights	15.1
	FP32 Adam moments	30.3
	Activations (one microbatch)	28.3
	Preserved prefix KV	16.0
Reference (TP=8)	BF16 weights	7.6
Rollout (TP=4)	BF16 weights	15.1
	KV cache	29.0
	Runtime buffers	2–4
Total (vs. 80 GB H800 HBM)		≈165

a more meaningful target than raw GPU utilization: a co-schedule is useful only to the extent that it hides required prefix work without expanding the decode window.

2.4 Challenges

Realizing this schedule creates two feasibility challenges: cross-phase state exceeds HBM, and rollout and training prefer different TP layouts.

2.4.1 Cross-Phase Memory Pressure. While overlap increases GPU utilization, it also introduces significant memory pressure. In sequential execution, rollout and training use HBM at different times, so only the active phase's model state and intermediate data need to be resident. Overlapping the phases instead requires data from multiple phases to coexist in limited HBM. Rollout KV must remain live during decode; reference and actor boundary KV must survive until suffix computation; training state must remain available for backward; and parameters, gradients, master weights, and optimizer moments are needed at different points in the update. Naively keeping these objects resident requires roughly 165 GiB per GPU (Table 1), more than twice the 80 GB capacity of an H800.

The problem is therefore not only allocation size but residency over time. A feasible system must retain latency-critical boundary KV, offload or recompute bulky autograd state, release rollout and scoring buffers at last use, and avoid materializing the full optimizer state at once. It must do so without severing the training-prefix autograd path that makes the reordered execution equivalent to the serial one.

2.4.2 Training/Rollout Layout Conflict. Second, rollout and training run in separate driver contexts and ordinarily allocate independent actor copies, adding roughly 15 GiB per GPU. Sharing one physical copy is complicated because the engines prefer different tensor-parallel (TP) degrees. Training uses a wider group, such as TP=8, to distribute parameters, activations, gradients, and optimizer state [31, 36]. Rollout prefers a narrower group, such as TP=4, because each transformer block places a collective on the per-token decode path [18, 28]. Matched TP=4 does not fit the training state, whereas matched TP=8 slows rollout by up to 1.31× (§4.5). An ideal system should therefore maintain one physical copy of the actor weights, share it across phases, and still allow rollout and training to use their preferred TP degrees.

Doing so introduces a further layout problem. Different TP degrees induce different shard boundaries, and Megatron and vLLM may order the same logical parameter differently in memory. A shared buffer is useful only if both engines can express their shards as valid views of the same physical bytes; otherwise the runtime must copy or permute the incompatible tensors.

Together, these constraints make naive spatial overlap impractical: it can exceed HBM capacity or force an inefficient common TP degree. Rollplex addresses both barriers without changing the synchronous update.

3 Rollplex Design

Rollplex realizes the cross-phase schedule of §2.3 with two mechanisms. *Phase-aware memory management* (§3.2) moves the iteration's changing working set through HBM without breaking the training graph. *Parallelism-aware weight sharing* (§3.3) lets training and rollout use different TP degrees while sharing compatible actor storage.

Rollplex is implemented in ROLL [39], with Megatron-Core 0.13.0 for training and vLLM 0.17.0 [18] for rollout. CUDA MPS provides cross-process execution concurrency, while CUDA VMM and IPC provide physical residency control and shared mappings.

3.1 Cross-Phase Execution

Rollplex colocates rollout and training as separate processes on one GPU pool (Figure 4). An orchestrator enforces phase barriers and treats rollout decode as the critical path. It leaves MPS limits unset; the resulting decode interference Δ_D is evaluated in §4.6. Prefix kernels run concurrently on separate process streams; if they outlast decode, their remainder stays on the critical path. Every overlapped prefix uses the same input and actor snapshot θ_k as the serial iteration, so the schedule changes only *when* independent work executes.

The orchestrator makes this dependence explicit with three barriers. During Phase 1, every engine treats θ_k as read-only. The Phase-2 suffixes start only after rollout has fixed the response tokens and their corresponding prefix boundaries are ready. The Phase-3 update starts only after reference scoring, actor forward, and backward have stopped reading θ_k. These barriers also delimit when physical pages

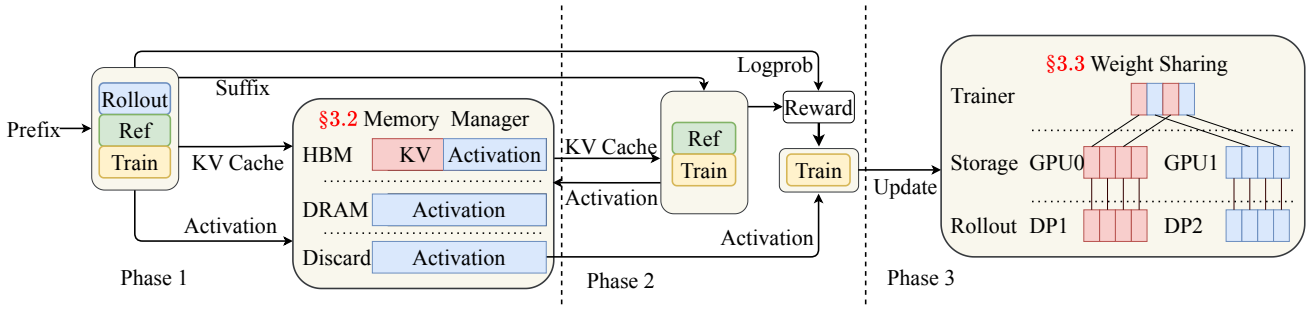

Figure 4. Rollplex architecture and one cross-phase iteration. Prefix work from reference scoring and actor training executes alongside rollout decode; response-dependent suffixes and backward follow generation.

may be released or rebound: an engine never observes a mapping change while one of its kernels can still dereference the old view.

Phase 1: rollout and prefix materialization. While rollout decodes responses under θ_k, the reference model and training actor encode the video and prefill the prompt. Each produces prefix log-probabilities and boundary KV state for its later suffix. Reference outputs are detached. The training boundary remains attached to autograd, while bulky saved prefix state is offloaded or marked for recomputation. During this window, the actor snapshot, rollout KV, and boundary state are hot in HBM; optimizer state is not.

Phase 2: suffix scoring and training. After generation, the reference and actor suffixes consume the responses and their preserved boundaries. Prefix and suffix log-probabilities form the full-sequence values used by the RL loss, after which the actor executes backward. Rollout KV and inference workspace are released before the training working set reaches its peak.

Phase 3: update and publication. The optimizer streams FP32 state through HBM and regenerates the BF16 actor snapshot chunk by chunk. Training ranks synchronize the updated pieces and populate any rollout-visible regions required by the sharing layout. Because compatible rollout tensors view the same IPC-backed storage, the completed snapshot is immediately visible for the next iteration; only incompatible tensors require reconstruction.

Let T_D be standalone decode time, T_P the makespan of the concurrently running reference and training prefixes, Δ_D the decode slowdown caused by co-running them, and T_{post} the response-dependent suffix, backward, and update time. Ignoring rollout prefill common to both schedules,

$$T_{\text{Rollplex}} = \max(T_D + \Delta_D,\ T_P) + T_{\text{post}},$$
$$S = T_D + T_P - \max(T_D + \Delta_D, T_P) = \min(T_P - \Delta_D, T_D). \quad (1)$$

Here S is the critical-path saving over the serial interval $T_D + T_P$. In the prefix-bounded regime, $S = T_P - \Delta_D$, so interference reduces the saving linearly; in the decode-bounded regime, $S = T_D$, and interference is hidden until the boundary. This decode-window cap explains why faster decode (§4.5) and rollout tail-cutting (§4.7) leave more prefix work on the critical path.

3.2 Phase-Aware Memory Management

Rollplex enforces phase-scoped memory residency with one invariant: an object occupies HBM only between its first latency-critical use and its last use. The relevant objects fall into four classes. *Permanent state*, such as FP32 master weights and Adam moments, may move between host and GPU but is never discarded. *Regenerable snapshots*, including BF16 actor weights, can be replaced after their last consumer finishes. *Boundary state*, principally prefix KV, remains resident until the corresponding suffix consumes it. *Phase-local state*, such as rollout KV, inference workspace, scoring temporaries, and checkpointed activations, is released at last use or recomputed.

Stable-address staging. Rollplex combines a CUDA-VMM allocator with a pinned host memory pool. VMM separates virtual tensor views from their physical backing: at a safe phase barrier, Rollplex can release or remap HBM pages without rebuilding the Megatron and vLLM tensor objects that refer to those address ranges [12, 22, 29]. VMM allocations are also exported through CUDA IPC so both processes can map compatible actor storage. The pinned host memory pool holds inactive permanent state and offloaded saved tensors; allocating these objects from one pool avoids fragmentation from many large pinned allocations.

The allocator reserves virtual ranges once and maps physical pages only while an object is resident. A release operation unmaps and frees the backing pages while retaining the reservation; remapping later attaches fresh pages at the same virtual address. The tensor's shape, strides, and storage offset therefore remain valid across an eviction. For shared actor ranges, the owner exports VMM allocation handles and the peer process imports and maps the same physical pages into its own address space. Imported mappings persist across iterations unless the underlying allocation is resized

or replaced. Rollplex preallocates the pinned host memory pool during initialization.

Using VMM and the preallocated host pool, Rollplex manages residency across the three execution phases. Rollout KV and inference buffers are released after generation. Boundary KV survives only until its Phase-2 suffix. Frozen reference weights are paged into HBM for their Phase-1 prefix and Phase-2 scoring windows and may be evicted between those windows. Training-prefix saved state is offloaded as it is produced or discarded for checkpoint recomputation. At the optimizer barrier, all consumers of the old BF16 snapshot have quiesced; Rollplex releases its physical pages and reuses that capacity as update workspace.

Consequently, the Phase-1 peak contains the actor snapshot, rollout KV, the two prefix boundaries, the reference weights only while their prefix runs, and small offload metadata. Suffix activations and gradients appear only in Phase 2, after rollout state is released. Full optimizer state never joins either peak; Phase 3 admits only a sliding window of it.

Chunked update. Loading FP32 master weights and Adam moments for the complete model would exceed HBM even after releasing the old BF16 snapshot. Rollplex therefore partitions parameters into byte-balanced chunks. For each chunk it admits the finalized gradient and corresponding FP32 state, applies Adam, casts the result to BF16, and writes the new snapshot into the rollout-visible destination selected by §3.3. Updated FP32 state is returned to the host, and the workspace is reused by the next chunk. Adjacent transfers and computation are pipelined when memory permits. Global loss-scale, norm, and clipping operations complete before any affected chunk is committed, so this changes residency rather than optimizer semantics.

The implementation uses separate load, update, and offload streams over adjacent chunks. While chunk j executes Adam, chunk $j+1$ may load its FP32 state and chunk $j-1$ may return its updated state to host. Chunk size is selected so this small pipeline, the partially regenerated BF16 snapshot, and the current gradient slice fit simultaneously. Peak update memory is therefore independent of the full FP32 optimizer footprint.

Training-prefix correctness. Offloading saved tensors does not detach the prefix computation. The actor prefix produces attached boundary KV; the suffix loss sends gradients through that boundary into the LLM prefix, projector, and vision encoder. Saved prefix tensors are reloaded or recomputed when backward needs them. Consequently, Rollplex preserves the serial autograd path while keeping only latency-critical boundary state in HBM across the decode window. The supplementary lifecycle table lists the complete per-object contract. The reference prefix has no autograd obligation and is detached; its retained boundary is consumed only by reference scoring.

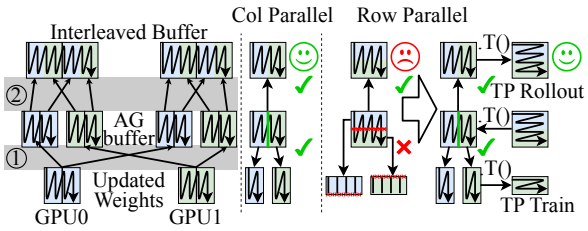

Figure 5. Weight sharing for $\mathrm{TP}_{\mathrm{train}} = 2\mathrm{TP}_{\mathrm{rollout}}$. **Left**: training ranks populate a rollout-sized shared region. **Middle**: same-axis shards alias directly. **Right**: row-parallel shards share a transposed storage order.

3.3 Parallelism-Aware Weight Sharing

Training and rollout require the same logical actor but prefer different TP degrees and may use different parameter orders. Rollplex chooses one VMM-backed physical placement for each compatible tensor, exports it through CUDA IPC, and binds each engine's local tensor to the appropriate region. When $\mathrm{TP}_{\mathrm{train}} > \mathrm{TP}_{\mathrm{rollout}}$, several training shards jointly cover one rollout shard; after an update, ranks exchange the pieces needed to populate each rollout-visible shared region.

Let $q = \mathrm{TP}_{\mathrm{train}}/\mathrm{TP}_{\mathrm{rollout}}$. The current sharing scheme applies when $q \in \mathbb{Z}_{>0}$. Rollplex groups every q colocated training ranks whose shards form one rollout shard. The rollout-sized IPC buffer is the actor snapshot itself rather than a separate post-update copy. Each training rank binds its parameter to the region it owns and writes its updated BF16 values there. A group-local exchange gathers the other $q-1$ pieces and scatters them to their predetermined offsets so that every rollout rank sees a complete shard. Layout classification determines whether this population is a direct placement, a metadata-compatible transpose, or an explicit reconstruction.

Sharing criterion. For logical element i of a tensor, let $\phi_E(i)$ denote the physical byte observed by engine E after metadata-only transforms accepted by its kernels. A tensor is zero-copy shareable when a common physical placement makes

$$\phi_{\mathrm{train}}(i) = \phi_{\mathrm{rollout}}(i) \quad \text{for every } i,$$

and each rank-local shard satisfies the engine's contiguity requirements. VMM operates on page-aligned imported ranges; tensor views select offsets within those ranges. Rollplex tests the criterion using the tensor's sharding axis, shard order, storage strides, and TP ratio, producing three cases.

This classification is performed once when the engines bind their parameters. Equality of the two mappings guarantees that a training write changes exactly the bytes later read by rollout. When no permitted metadata transform makes the mappings equal, Rollplex marks the tensor for reconstruction rather than exposing an incorrect alias.

Case 1: identical or nested sharding. If both engines shard the same axis in the same order, every training shard is a contiguous region of the rollout placement. This includes identical TP and nested partitions in which the wider training TP subdivides a rollout shard, as well as replicated vision parameters. Both engines bind views of the common allocation; no layout conversion is required.

Case 2: transposed storage. Some row-parallel weights fail Case 1 only because the sharded axis is innermost in row-major storage. Rollplex stores such a weight in the transposed order, making each shard a contiguous slab, and exposes the expected logical orientation through tensor metadata. The physical bytes remain shared; GEMM kernels use their transpose mode. Concretely, a logical $[O, I]$ matrix sharded along I is stored as $[I, O]$ so every training and rollout slice occupies a contiguous range; neither engine materializes the logical transpose.

Case 3: incompatible permutation. A non-affine permutation cannot satisfy the common-placement criterion. For example, vLLM and Megatron can order fused-QKV heads differently (interleaved versus grouped). The reordered fragments are smaller than the 2 MB VMM page granularity and cannot be reconciled by page remapping or a single strided view. Rollplex therefore keeps a separate inference-layout buffer only for these tensors and copies and permutes them after an update.

Binding and refresh. The streaming update writes Case-1 and Case-2 BF16 values into shared destinations and reconstructs only Case 3; Case 2 uses the common transposed order. Case-3 reconstruction starts after its training chunk is final and finishes before rollout binds the new snapshot. The Phase-3 barrier prevents either engine from mixing chunks from θ_k and θ_{k+1}. Mappings persist across iterations; before remapping or resizing a buffer, the orchestrator quiesces both engines. This avoids a second actor while preserving each engine's preferred TP degree.

4 Evaluation

We evaluate Rollplex by answering five questions: **Q1:** How much does prefix–decode overlap reduce end-to-end step time over colocated and disaggregated deployments? (§4.2) **Q2:** Does the new schedule change what is learned? (§4.3) **Q3:** Are Rollplex's two techniques necessary for the overlap: does phase-aware memory management keep the schedule within HBM (§4.4), and how much performance does parallelism-aware weight sharing provide (§4.5)? **Q4:** How should the co-running processes share GPU execution resources? (§4.6) **Q5:** Does Rollplex remain effective on top of asynchronous rollout-side optimizations? (§4.7)

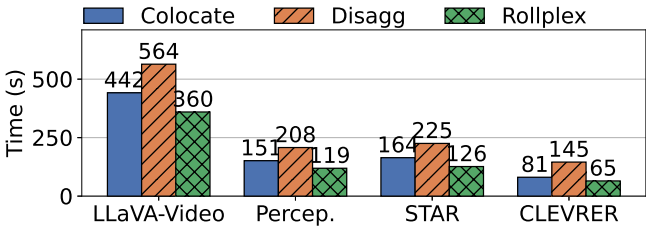

Figure 6. Per-step execution time of Colocate, Disagg, and Rollplex on Qwen2.5-VL-32 B over 32×H800.

4.1 Experimental Setup

Models. Unless otherwise stated, we train Qwen2.5-VL-32 B [2]. All training follows Video-R1's reinforcement learning with verifiable rewards (RLVR) setup [9], using Group Relative Policy Optimization (GRPO) [32] with $N = 8$ responses per prompt group. Unless otherwise stated, the maximum sequence length is 10k tokens, split into an 8k-token prompt budget and a 2k-token response budget, and the rollout batch size is 32 prompts.

Cluster Setup. We evaluate Rollplex on a 4-node H800 cluster with 32 GPUs in total, 1.4 TB of host memory per node, and 400 Gbps InfiniBand. The memory microbenchmarks use a single 8-GPU H800 node. The resource-policy study of §4.6 uses 8×H20 (78 SMs per GPU).

Datasets. We use CLEVRER [44], PerceptionTest [27], LLaVA-Video-178K [50], and STAR [40]. These tasks span short synthetic physical reasoning clips, real-world video understanding, long-video question answering, and situated action reasoning.

Measurement. After discarding the first iteration for engine warmup and CUDA-graph capture, we report average time per step over the remaining iterations.

4.2 End-to-End Evaluation

We compare the end-to-end performance of Rollplex with two major deployment paradigms:

1. **Colocate.** Rollout and training share the same GPU pool, but phases execute sequentially, following the common colocated ROLL execution style [39].
2. **Disagg.** Rollout uses two dedicated 8-GPU nodes with `TP=4`, while training uses two dedicated 8-GPU nodes with `TP=8`. This preserves the same 32-GPU budget and requires cross-pool actor-weight synchronization after each update.

We implement every system in ROLL [39]. Each enables the same prefix-sharing optimization [20, 38, 48]. All systems use the same model, dataset order, reward function, rollout batch size, and maximum sequence lengths.

As shown in Figure 6, Rollplex delivers the lowest step time on every dataset. It reduces step time by 1.23×–1.30× over Colocate and by 1.57×–2.24× over Disagg. The gain over

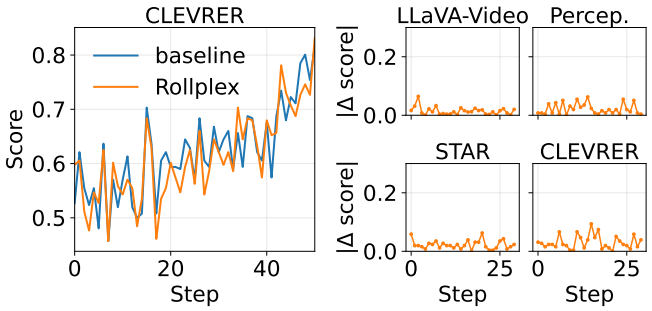

Figure 7. Training-quality comparison. **Left**: Verifiable reward per step for Rollplex and the colocated baseline on CLEVRER. **Right**: Absolute per-step reward difference between the two systems on all four datasets.

Colocate comes from moving response-agnostic prefix computation into the rollout-decode window, while Rollplex's memory and weight-sharing mechanisms keep the overlapped working set within HBM.

The gain is consistent across datasets, and its magnitude reflects the decode-window cap of §2.3: overlap can hide at most the shorter of decode time and total prefix time, while the response-dependent suffixes, the backward pass, and the model update remain on the critical path. Hiding the prefix work alone therefore yields around 1.2×–1.3× end-to-end.

Disagg trails Colocate on every dataset despite using the same 32-GPU budget: with dedicated pools, the rollout pool idles during scoring and training while the training pool idles during rollout, and per-iteration cross-pool weight synchronization adds further overhead. The cost is proportionally largest on CLEVRER (1.80×), whose short iterations amortize these fixed costs least.

4.3 End-to-End Reward Comparison

The speedup of §4.2 is only meaningful if training quality is unchanged. The systems techniques in Rollplex change when computation runs, not the GRPO update being computed: the overlapped prefixes are computed under the same on-policy snapshot θ_k, and the preserved prefix KV is exactly the state the serial schedule would recompute. To confirm this empirically, we train with Rollplex and the colocated baseline under the same model, data order, reward function, and response budget, and compare the per-step verifiable reward of the two systems on all four datasets.

Exact trajectories need not match because GRPO training is stochastic, and different GPU kernel choices can introduce small floating-point differences. Nevertheless, Figure 7 left shows that the two CLEVRER reward curves closely track each other throughout training and reach similar final rewards. Figure 7 right plots the absolute per-step reward difference between the two systems on all four datasets. It stays near zero across training, with no growth or drift. This negligible reward gap is within the normal step-to-step variation of GRPO sampling. Rollplex therefore improves time-to-train without affecting training quality.

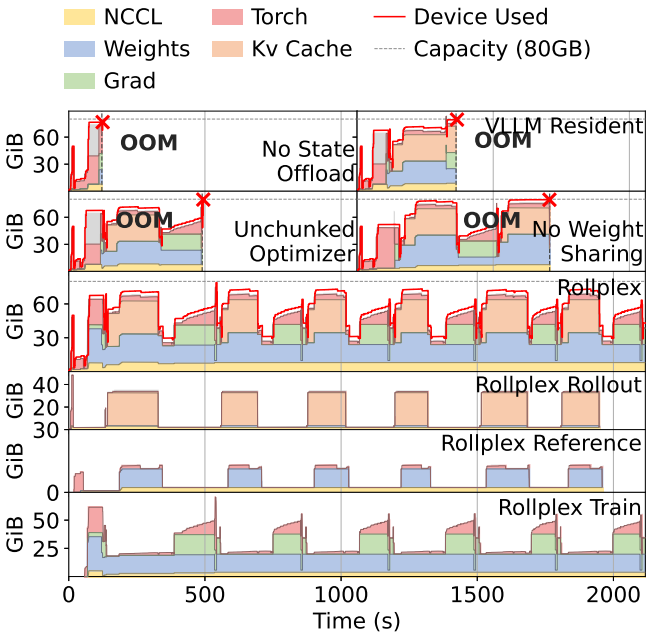

Figure 8. Per-GPU memory over time. Top: four ablations; middle: full Rollplex; bottom: per-role breakdown. Dashed line: 80 GB capacity.

4.4 Memory Feasibility

To fit in HBM, the overlapped schedule uses phase-aware memory management (§3.2). On the 8×H800 microbenchmark with the 32 B model, we profile per-GPU memory for full Rollplex and four ablations: *no state offload* keeps optimizer state and activations in HBM; *vLLM resident* retains rollout KV and inference buffers; *unchunked optimizer* loads all FP32 master weights and Adam state together; and *no weight sharing* duplicates the actor weights.

Figure 8 shows the result: all four ablations run out of memory, while full Rollplex runs continuously with its peak below the 80 GB capacity. The failures occur where the lifetime analysis of §2.4.1 predicts. Without state offload, the working set exceeds capacity already in the first iteration. The remaining three ablations all die at an optimizer step, the peak-memory point of the iteration: the unchunked optimizer loads the full FP32 master weights and Adam state at once and fails immediately. Keeping vLLM state resident or duplicating the actor weights survives longer, but the extra resident state eventually collides with the optimizer's transient peak. Rollplex's chunked optimizer caps exactly this peak, which, combined with releasing the BF16 weight snapshot at the barrier, is what keeps the full system under capacity.

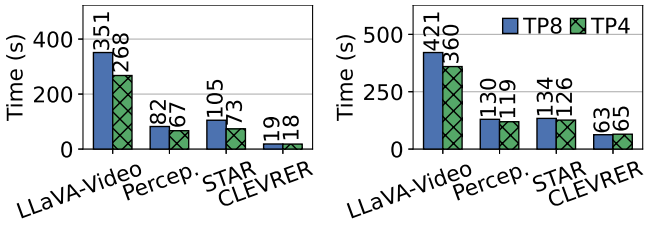

Figure 9. Rollout TP=8 versus TP=4 under Rollplex with training fixed at TP=8. *Left:* rollout time. *Right:* per-step time.

The per-role panels illustrate phase-scoped memory residency. Rollout KV occupies HBM only during the decode windows and is released after generation. Reference weights are loaded around reference compute and offloaded as soon as it completes. On the training process, the BF16 actor snapshot is released at the optimizer barrier, which appears as the dip in the weights band, so that the streamed optimizer chunks can reuse that memory before the next snapshot is written back. Finally, the gradual within-iteration slope in the stacked areas comes from the PyTorch caching allocator, which retains freed blocks rather than returning them to the device. This is reused cache, not live tensor state, and does not reflect actual memory demand.

Weight-layout coverage. For Qwen2.5-VL-32 B, Cases 1/2/3 cover 19.7/10.7/2.4 B parameters (60.0/32.8/7.2%). Cases 1 and 2 therefore share physical storage for 92.8% of parameters. The 1.1 GiB Case-3 buffer per GPU at rollout TP=4 is 85.5% smaller than duplicating the smaller TP=8 actor shard (7.6 GiB; Table 1).

4.5 Tensor-Parallel Degree versus End-to-End Time

We evaluate parallelism-aware weight sharing (§3.3) by comparing rollout at TP=8, which matches training, with its preferred TP=4. Both use the 32×H800 setup of §4.2, with training fixed at TP=8.

Figure 9 (left) shows the cost of the matched degree. On LLaVA-Video, PerceptionTest, and STAR, TP=8 slows rollout by 1.17×–1.31× relative to TP=4, because the all-reduce on the per-token decode path spans twice as many ranks. On CLEVRER, whose rollouts are too short for the communication penalty to accumulate, the two degrees are within a few percent. The end-to-end picture (right) follows: rollout TP=4 reduces per-step time by 1.06×–1.17× on the three longer datasets, while on CLEVRER the two configurations again differ by only a few percent. The end-to-end gap is also smaller than the rollout gap because part of the rollout saving is reabsorbed by the schedule: a faster decode window hides less prefix work, so some previously overlapped prefix computation returns to the critical path. This is most visible on STAR, where a 25-second rollout saving yields a 7-second step-time saving.

Table 2. Resource-sharing policies for Qwen2.5-VL-7B on 8×H20. Each cell reports total step time (top) and rollout generation time (bottom), in seconds. †Sequential disables cross-phase overlap and provides the standalone rollout reference.

Policy	CLEVRER	LLaVA	Percep.	STAR
Sequential (no overlap)†	44.5 15.0	216.1 133.7	163.4 79.7	56.2 23.9
MPS-Default	39.2 16.7	**197.8** 134.9	115.5 81.4	41.1 23.8
ThreadCap 80%/20%	40.0 19.0	199.6 135.4	120.8 84.5	**40.7** 23.6
ThreadCap 70%/30%	126.6 73.8	259.5 177.0	222.6 150.1	91.6 65.0
SM-Affinity 80%/20%	**38.2** 17.3	200.2 137.3	115.4 76.3	42.2 24.3
SM-Affinity 70%/30%	131.2 75.5	259.5 175.7	213.5 140.4	93.7 65.5
GreenCtx 62/16 SMs	42.4 24.3	201.0 138.3	**114.6** 79.7	43.5 27.4
GreenCtx 54/24 SMs	42.3 22.6	203.0 140.6	115.8 79.9	45.8 27.9
GreenCtx 38/40 SMs	43.6 23.1	199.0 137.2	119.0 82.0	45.5 27.6

Letting rollout keep its preferred degree helps wherever rollout is long enough to matter and costs nothing elsewhere. Parallelism-aware weight sharing makes this choice free: rollout runs at TP=4 against the TP=8 training layout while both engines read one physical actor weight allocation, with no duplicate actor copy or separate full-model layout conversion. Without it, a colocated deployment must either accept the matched-TP slowdown or pay the duplicate-weight memory cost of §2.4.2.

4.6 Resource Sharing Policy

The overlap model of Equation 1 depends on keeping rollout interference Δ_D small. We compare default MPS with explicit MPS limits and GreenCtx partitions in a single-node setup that isolates GPU-sharing interference from cross-node communication. The setup uses Qwen2.5-VL-7B on 8×H20, with Megatron training at TP=4 and two vLLM rollout clients at TP=4. *ThreadCap* sets per-client active-thread percentages; *SM-Affinity* converts them to SM-count affinity. MPS ratios list training followed by each rollout client. *GreenCtx-Partition* reports actual training/rollout SM counts and runs without MPS. All configurations use the same overlapped schedule.

Work-conserving overlap keeps decode interference small. Relative to standalone rollout under sequential execution, MPS-Default adds at most 1.7 seconds of generation time. This is 0.9% on LLaVA and 2.1% on PerceptionTest; short CLEVRER incurs 11.3% but only 1.7 seconds in absolute

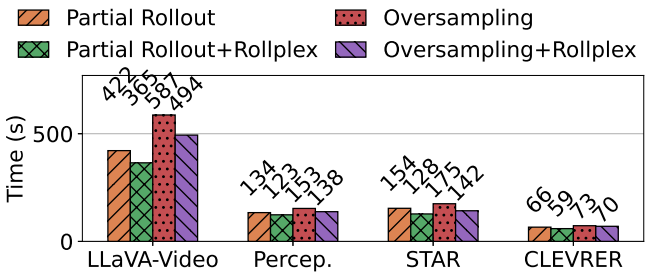

Figure 10. Per-step time of partial rollout and oversampling, each with and without Rollplex.

terms, while STAR differs by 0.1 seconds within measurement noise. The overlap nevertheless reduces total step time by 1.09×–1.41×.

Default MPS is robust without tuning. MPS-Default remains within 2.7% of the best policy on every dataset without tuning because either process can consume idle capacity.

Rigid limits cause a performance cliff. Both 80%/20% MPS policies remain within 5% of MPS-Default, but changing the limits to 70%/30% increases total time by 1.30×–3.44× relative to the corresponding 80%/20% policy. This nonlinear response is consistent with ZipBatch's observation that rigid GPU allocations poorly match kernels whose compute and bandwidth demands vary over time [46, 51]. Fixed compute limits do not isolate shared HBM bandwidth, so they can throttle prefix progress while retaining decode contention. MPS-Default instead lets either process use idle capacity.

Static GreenCtx partitions do not consistently improve the joint critical path. GreenCtx executes successfully by itself, but its total time ranges from 0.8% better to 11.4% worse than MPS-Default and it slows generation most visibly on the short CLEVRER and STAR workloads. Because its SM groups are static, capacity left idle by one process cannot be borrowed by another. We therefore use MPS-Default: it is not the lowest-time cell for every workload, but it stays close to the best without a workload-specific partition.

4.7 Composition with Asynchronous Rollout Optimizations

On the same 32 B/32×H800 setup as §4.2, Figure 10 shows that Rollplex's benefit carries over to partial rollout and oversampling [11, 55]. Applied on top, it reduces per-step time by 1.08×–1.21× for partial rollout and 1.04×–1.23× for oversampling. Absolute step times are not comparable across methods because each changes the amount of rollout work per step, so we compare each method only with and without Rollplex. The gain is slightly smaller than over the plain colocated baseline because tail-cutting shortens the decode window and leaves less room to hide prefix work.

5 Discussion

Applicability Envelope. Rollplex applies when substantial training-side work is a *response-agnostic prefix* that depends on the prompt and snapshot θ_k, but not the rollout response. VLM RL fits this regime because video encoding and visual-token prefill can materialize prefix KV before the response is known (§2.1). Benefits are largest when prefix work rivals decode and the GPU budget cannot support separate rollout and training pools at their preferred TP degrees; they diminish when the prefix is small or a clean split is affordable.

Embodied RL. VLMs ground agents visually, while vision-language-action models map observations and instructions to robot actions [3, 6, 17]. Their trajectories create long visual prefixes from observations, goals, and feedback. When a fixed history is response independent, Rollplex can harvest decode slack as in our VLM workloads. Closed-loop deployment additionally requires trajectory-aware KV ownership because later observations depend on earlier actions.

Fault Tolerance and Blast Radius. Rollplex relies on soft GPU sharing through separate CUDA contexts, shared IPC mappings, streams, and MPS. These mechanisms multiplex kernels and memory but do not provide independent recovery boundaries. If either colocated engine triggers a fatal device error, invalidates an IPC mapping, or wedges a collective, the shared pool will likely lose the iteration. Disaggregation can instead place the engines in separate pools, enabling independent restart and failure containment. This limitation is orthogonal to Rollplex's overlap schedule and memory-lifecycle design; we leave fault isolation and recovery for future work.

6 Related Work

6.1 RL Post-Training Execution

LLM RL systems orchestrate distributed training and generation [16, 33, 35, 39, 42, 57]. They use temporal colocation [35, 39, 54], disaggregation [16, 41, 43, 53], or asynchronous execution that relaxes rollout–update ordering [10, 49].

Rollplex instead spatially overlaps independent work from the *same* synchronous iteration. Unlike training systems that pipeline, co-schedule, or rematerialize work within training [7, 14, 19], it crosses rollout and training engines. It also lets resident engines read one snapshot at different TP degrees rather than converting layouts or transferring snapshots [35, 43].

6.2 GPU Spatial Sharing

NVIDIA MPS overlaps kernels and memory copies across processes; MIG creates isolated compute/memory partitions [24, 25]. Neither chooses co-runners [4]. Research systems add memory sharing [47], preemption [13], profile-guided co-scheduling [37], compilation-aware scheduling [21], or dynamic compute-and-bandwidth regulation [46, 51].

Prior systems multiplex independent jobs for isolation, fairness, or throughput. Rollplex uses MPS but derives co-runners from the on-policy dependence graph and coordinates autograd state, KV caches, optimizer residency, and weights shared across training and serving. NanoFlow overlaps operations inside one serving engine [56]; Rollplex instead overlaps models and engines across RL phases.

BEEMS smooths vision-inference memory demand, while vAttention and GMLake use VMM for stable KV addresses and reduced fragmentation [12, 29, 34]. Rollplex instead uses VMM for phase-scoped memory residency and cross-process weight views.

6.3 Rollout Underutilization

APRIL [55] and RollPacker [11] fill decode slack with more rollout work. Serving systems instead co-schedule prefill with decode [1, 45] or disaggregate the two [8, 15, 23, 26, 30, 52], using other requests to improve serving. Rollplex instead fills the window with response-independent reference and training prefixes from the same iteration, the dominant exposed cost in VLM RL (§2.2), without changing sampled responses or the PPO/GRPO data dependence.

6.4 Prefix Sharing

Prefix sharing [20, 38, 48] removes redundant prompt-side computation across trajectories sharing a prompt, as in GRPO, where the N sampled responses in a group share the same video-text prefix. Rollplex implements the same semantics in all baselines: each role computes its GRPO prefix once per prompt group and reuses it across the sampled responses. Thus, our speedups do not come from eliminating redundant prefix recomputation in the baselines, but from moving the already-deduplicated, response-independent prefix work into the rollout-decode window and preserving its KV state for later scoring and training.

7 Conclusion

In this paper, we present Rollplex, a runtime that overlaps VLM prefixes with rollout decode via phase-aware memory management and parallelism-aware weight sharing. On 32×H800 GPUs, it speeds up serial colocation by 1.23×–1.30× and disaggregation by 1.57×–2.24× under the same GPU budget.

Acknowledgments The authors used ChatGPT/Codex for paper refinement and figure generation. All data are collected and verified by human authors. All text and figures are verified by human authors.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. 2025. Qwen2.5-VL Technical Report. arXiv:2502.13923 [cs.CV] https://arxiv.org/abs/2502.13923

[3] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, Pete Florence, Chuyuan Fu, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Kehang Han, Karol Hausman, Alexander Herzog, Jasmine Hsu, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Lisa Lee, Tsang-Wei Edward Lee, Sergey Levine, Yao Lu, Henryk Michalewski, Igor Mordatch, Karl Pertsch, Kanishka Rao, Krista Reymann, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Pierre Sermanet, Jaspiar Singh, Anikait Singh, Radu Soricut, Huong Tran, Vincent Vanhoucke, Quan Vuong, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Jialin Wu, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. 2023. RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. arXiv:2307.15818 [cs.RO] https://arxiv.org/abs/2307.15818

[4] Chao Chen, Chris Porter, and Santosh Pande. 2022. CASE: A Compiler-Assisted Scheduling Framework for Multi-GPU Systems. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, 17–31. doi:10.1145/3503221.3508423

[5] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=H4DqfPSibmx

[6] Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. 2023. PaLM-E: An Embodied Multimodal Language Model. arXiv:2303.03378 [cs.LG] https://arxiv.org/abs/2303.03378

[7] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery, 431–445. doi:10.1145/3437801.3441593

[8] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1283–1295. doi:10.1145/3695053.3730999

[9] Kaituo Feng, Kaixiong Gong, Bohao Li, Zonghao Guo, Yibing Wang, Tianshuo Peng, Junfei Wu, Xiaoying Zhang, Benyou Wang, and Xiangyu Yue. 2025. Video-R1: Reinforcing Video Reasoning in MLLMs. arXiv:2503.21776 [cs.CV] https://arxiv.org/abs/2503.21776

[10] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. 2026. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning. arXiv:2505.24298 [cs.LG] https://arxiv.org/abs/2505.24298

[11] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. 2026. RollPacker: Taming Long-Tail Rollouts for RL Post-Training with Tail Batching. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 849–866. https://www.usenix.org/conference/nsdi26/presentation/gao-wei

[12] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 450–466. doi:10.1145/3620665.3640423

[13] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[14] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, 120–134. doi:10.1145/3503221.3508418

[15] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, Yun Liang, and Yu Wang. 2025. semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage. arXiv:2504.19867 [cs.CL] https://arxiv.org/abs/2504.19867

[16] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Weikai Fang, Xianyu, Yu Cao, Haotian Xu, and Yiming Liu. 2025. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. arXiv:2405.11143 [cs.AI] https://arxiv.org/abs/2405.11143

[17] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. 2024. OpenVLA: An Open-Source Vision-Language-Action Model. arXiv:2406.09246 [cs.RO] https://arxiv.org/abs/2406.09246

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[19] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. 2025. Mario: Near Zero-Cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25)*. Association for Computing Machinery, 197–211. doi:10.1145/3710848.3710878

[20] Zikang Liu, Tongtian Yue, Yepeng Tang, Longteng Guo, Junxian Cai, Qingbin Liu, Xi Chen, and Jing Liu. 2025. Prefix Grouper: Efficient GRPO Training through Shared-Prefix Forward. arXiv:2506.05433 [cs.LG] https://arxiv.org/abs/2506.05433

[21] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 595–610. doi:10.1145/3600006.3613163

[22] NVIDIA. 2020. Introducing Low-Level GPU Virtual Memory Management. https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/. NVIDIA Developer Blog.

[23] NVIDIA. 2025. NVIDIA Dynamo: A Datacenter-Scale Distributed Inference Serving Framework. https://github.com/ai-dynamo/dynamo.

[24] NVIDIA. 2026. CUDA Multi-Process Service. https://docs.nvidia.com/deploy/mps/latest/index.html. NVIDIA documentation, accessed July 2026.

[25] NVIDIA. 2026. Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. NVIDIA documentation, accessed July 2026.

[26] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture* (Buenos Aires, Argentina) *(ISCA '24)*. IEEE Press, 118–132. doi:10.1109/ISCA59077.2024.00019

[27] Viorica Patraucean, Lucas Smaira, Ankush Gupta, Adria Recasens Continente, Larisa Markeeva, Dylan Sunil Banarse, Skanda Koppula, Joseph Heyward, Mateusz Malinowski, Yi Yang, Carl Doersch, Tatiana Matejovicova, Yury Sulsky, Antoine Miech, Alexandre Fréchette, Hanna Klimczak, Raphael Koster, Junlin Zhang, Stephanie Winkler, Yusuf Aytar, Simon Osindero, Dima Damen, Andrew Zisserman, and Joao Carreira. 2023. Perception Test: A Diagnostic Benchmark for Multimodal Video Models. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=HYEGXFnPoq

[28] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. arXiv:2211.05102 [cs.LG] https://arxiv.org/abs/2211.05102

[29] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. arXiv:2405.04437 [cs.LG] https://arxiv.org/abs/2405.04437

[30] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[32] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300

[33] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, Markel Sanz Ausin, Ashwath Aithal, and Oleksii Kuchaiev. 2024. NeMo-Aligner: Scalable Toolkit for Efficient Model Alignment. arXiv:2405.01481 [cs.CL] https://arxiv.org/abs/2405.01481

[34] Hanjing Shen, Fangxin Liu, Jian Liu, Li Jiang, and Haibing Guan. 2026. BEEMS: Boosting Machine Vision Efficiency via Computation Graph-Based Memory Smoothing. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26)*. Association for Computing Machinery, 496–508. doi:10.1145/3774934.3786430

[35] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1279–1297. doi:10.1145/3689031.3696075

[36] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. (2020). arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[37] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. doi:10.1145/3627703.3629578

[38] Jinghui Wang, Shaojie Wang, Yinghan Cui, Xuxing Chen, Chao Wang, Liang Huang, Can Tang, Xiaojiang Zhang, Junyi Peng, Li Wan, Haotian Zhang, and Bin Chen. 2026. Tree Training: Accelerating Agentic LLMs Training via Shared Prefix Reuse. arXiv:2511.00413 [cs.LG] https://arxiv.org/abs/2511.00413

[39] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang, and Bo Zheng. 2025. Reinforcement Learning Optimization for Large-Scale Learning: An Efficient and User-Friendly Scaling Library. *arXiv preprint arXiv:2506.06122* (2025).

[40] Bo Wu, Shoubin Yu, Zhenfang Chen, Joshua B. Tenenbaum, and Chuang Gan. 2021. STAR: A Benchmark for Situated Reasoning in Real-World Videos. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. https://openreview.net/forum?id=EfgNF5-ZAjM

[41] Tianyuan Wu, Lunxi Cao, Yining Wei, Wei Gao, Yuheng Zhao, Dakai An, Shaopan Xiong, Zhiqiang Lv, Ju Huang, Siran Yang, Yinghao Yu, Jiamang Wang, Lin Qu, and Wei Wang. 2025. RollMux: Phase-Level Multiplexing for Disaggregated RL Post-Training. arXiv:2512.11306 [cs.DC] https://arxiv.org/abs/2512.11306

[42] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai Che, Shuaiwen Leon Song, and Yuxiong He. 2023. DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales. arXiv:2308.01320 [cs.LG] https://arxiv.org/abs/2308.01320

[43] Chenhao Ye, Huaizheng Zhang, Mingcong Han, Baoquan Zhong, Xiang Li, Qixiang Chen, Xinyi Zhang, Weidong Zhang, Kaihua Jiang, Wang Zhang, He Sun, Wencong Xiao, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2026. TensorHub: Scalable and Elastic Weight Transfer for LLM RL Training. arXiv:2604.09107 [cs.DC] https://arxiv.org/abs/2604.09107

[44] Kexin Yi, Chuang Gan, Yunzhu Li, Pushmeet Kohli, Jiajun Wu, Antonio Torralba, and Joshua B. Tenenbaum. 2020. CLEVRER: Collision Events for Video Representation and Reasoning. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkxYzANYDB

[45] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for

Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[46] Haoxuan Yu, Sheng Yao, and Wei Wang. 2025. ZipBatch: Multi-Tenant GPU Batching with Dual-Resource Regulation. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 240–254. doi:10.1145/3772052.3772229

[47] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 98–111. https://proceedings.mlsys.org/paper_files/paper/2020/file/d9cd83bc91b8c36a0c7c0fcca59228f2-Paper.pdf

[48] Jiarui Zhang, Yuchen Yang, Ran Yan, Zhiyu Mei, Liyuan Zhang, Daifeng Li, Wei Fu, Jiaxuan Gao, Shusheng Xu, Yi Wu, and Binhang Yuan. 2026. AREAL-DTA: Dynamic Tree Attention for Efficient Reinforcement Learning of Large Language Models. arXiv:2602.00482 [cs.LG] https://arxiv.org/abs/2602.00482

[49] Liujie Zhang, Benzhe Ning, Rui Yang, Xiaoyan Yu, Jiaxing Li, Lumeng Wu, Jia Liu, Minghao Li, Weihang Chen, Weiqi Hu, and Lei Zhang. 2026. Relax: An Asynchronous Reinforcement Learning Engine for Omni-Modal Post-Training at Scale. arXiv:2604.11554 [cs.CL] https://arxiv.org/abs/2604.11554

[50] Yuanhan Zhang, Jinming Wu, Wei Li, Bo Li, Zejun Ma, Ziwei Liu, and Chunyuan Li. 2025. LLaVA-Video: Video Instruction Tuning With Synthetic Data. *Transactions on Machine Learning Research* (2025). https://openreview.net/forum?id=EElFGvt39K

[51] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yunzhe Li, Zhifeng Jiang, Yang Li, Xiaowen Chu, and Huaicheng Li. 2025. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25)*. Association for Computing Machinery, 267–281. doi:10.1145/3710848.3710863

[52] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[53] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. 2025. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation. arXiv:2504.15930 [cs.LG] https://arxiv.org/abs/2504.15930

[54] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. 2025. Optimizing RLHF Training for Large Language Models with Stage Fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 489–503. https://www.usenix.org/conference/nsdi25/presentation/zhong

[55] Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, Kangrui Du, Jialian Wu, Ximeng Sun, Jiang Liu, Qiaolin Yu, Hao Chen, Zicheng Liu, and Emad Barsoum. 2025. APRIL: Active Partial Rollouts in Reinforcement Learning to Tame Long-tail Generation. arXiv:2509.18521 [cs.LG] https://arxiv.org/abs/2509.18521

[56] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2025. NanoFlow: Towards Optimal Large Language Model Serving Throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 749–765. https://www.usenix.org/conference/osdi25/presentation/zhu-kan

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

A Appendix

This appendix contains detailed implementation contracts that are useful for reproducing Rollplex but are not necessary for following the main execution schedule.

A.1 Intermediate-data lifecycle

Table 3 gives the complete per-object state contract behind the lifecycle policy of §3.2: every tensor class whose lifetime spans a phase boundary of the Rollplex iteration, where it is produced and consumed, whether it is retained on the autograd graph, and where it resides in the memory hierarchy. The *Produced*/*Consumed* columns define each object's lifetime against the three-phase schedule of §3.1; *Autograd* records whether the object must stay attached to the backward graph for gradient correctness; *Residency policy* states whether it is pinned in HBM, offloaded to the host-side pinned pool, or streamed in chunks. The final column marks the objects that are simultaneously live during the Phase 1 overlap window and therefore determine whether colocated rollout and prefix computation fit within HBM.

The rows fall into four groups. First, *inference-side state* (rollout KV) is HBM-resident only while generation runs and carries no autograd obligation. Second, *boundary state* (the two boundary prefix KVs and the prefix logprobs) is the small, latency-critical output of the Phase 1 prefix forwards that Phase 2 consumes; it stays in HBM across the phase boundary, and only the training actor's copy is attached, since gradients re-enter the prefix through that KV boundary. Third, *bulky training state* (training-prefix activation and checkpoint state, the gradient buffer, and FP32 master weights with Adam moments) is exactly what Rollplex refuses to keep resident: activation state is moved out of HBM the moment it is produced, by offloading to the pinned pool or by discarding for recomputation (§3.2), and rematerialized for the Phase 2 backward, gradients are offloaded as chunks finalize, and optimizer state is streamed one chunk at a time during Phase 3. Fourth, *regenerable or frozen state* (the BF16 actor snapshot and the reference-model parameters) carries no autograd obligation and can be discarded and rebuilt—the old snapshot is discarded at the optimizer barrier and the next one regenerated chunk by chunk from the FP32 master weights during the Phase 3 stream, and the frozen reference weights are paged into HBM only around their Phase 1 prefix and Phase 2 scoring windows.

The final column then makes the overlap-feasibility argument of §3.2 concrete: the Phase 1 peak comprises only rollout KV, the boundary KVs, the shared actor snapshot, the transiently resident reference weights, and small residuals (prefix logprobs and offload handles)—while suffix activations, gradients, and optimizer state contribute nothing to the Phase 1 budget. This is the per-object justification for why the combined working set fits where the naive colocation of §2.4 does not.

Table 3. Intermediate-data state contract for one Rollplex iteration (P1/P2/P3 = the three phases of §3.1). *Autograd* states whether the object is retained on the Megatron/PyTorch backward graph; *Residency* gives the phase-scoped policy. The rightmost column marks the objects that count toward the Phase 1 overlap-window peak. Phase 2/3-only state does not inflate the Phase 1 budget. The training-actor prefix produces attached boundary KV and activation state. The activation state is offloaded immediately and rematerialized or recomputed for backward, which preserves the prefix backward path without keeping the full forward tape resident in HBM.

Object	Produced	Consumed	Autograd	Residency policy	In P1 peak
Rollout KV	P1 prefill	P1 decode	n/a (inference)	HBM; released after generation	yes
Boundary prefix KV (training actor)	P1 prefix fwd	P2 suffix fwd/bwd	**attached**	HBM; kept until suffix consumes it	yes
Boundary prefix KV (reference)	P1 prefix fwd	P2 scoring	detached	HBM; kept until scoring consumes it	yes
Reference-model parameters	permanent (frozen)	P1 ref prefix / P2 ref scoring	n/a (frozen)	PinnedPool; loaded to HBM around reference compute, offloaded when it completes	yes (during ref prefix)
Prefix logprobs	P1 prefix fwd	P2 scoring/-training	detached or attached by role	HBM or host; retained until combined with suffix logprobs	yes (small)
Training-prefix activation/checkpoint state	P1 prefix fwd	P2 backward (reload or recompute)	attached	Offloaded immediately to PinnedPool or dropped for recompute; rematerialized during backward	yes (handles)
Suffix activations	P2 suffix fwd	P2 backward	attached	HBM (checkpointed); P2 working set	P2-only
Gradient buffer	P2 backward	P3 optimizer	leaf grads	Offloaded to host as chunks finalize; readmitted per chunk in P3	no
BF16 actor snapshot θ_k	P3 cast (prev iter)	P1/P2 fwd+bwd	n/a (regenerable)	HBM; old snapshot discarded at optimizer barrier, regenerated from FP32	yes
FP32 master weights + Adam moments	permanent	P3 optimizer	n/a	PinnedPool; streamed one chunk at a time in P3	no