Tree Training: Efficient LLM Training on Tree-Structured Trajectories

Jinghui Wang **Shaojie Wang** **Can Tang** **Yinghan Cui**

Xuxing Chen **Xiaojiang Zhang** **Chao Wang** **Haotian Zhang**

Abstract

Whenever LLM training involves multiple outputs from the same input—agentic multi-turn trajectories, self-consistency reasoning, beam search distillation, or multi-rollout SFT/RL—the tokens form a tree-structured trajectory with shared prefixes. Existing pipelines linearize such data and treat each branch independently, causing substantial redundant computation that scales with the number of samples. We derive that averaging the loss over all branches is algebraically identical to a per-token weighted loss, reducing the problem to computing each token's log-probability exactly once. We propose DFS serialization of the tree, which visits every token exactly once, and adapt full-attention and SSM layers to ensure the resulting log-probabilities match independent per-branch computation exactly. For memory-constrained settings where the full tree exceeds GPU capacity, we propose **Redundancy-Free Tree Partitioning**, which achieves zero redundant computation with peak memory bounded by a single root-to-leaf path. Together, these contributions form **Tree Training**, achieving up to **6.2**$\times$ end-to-end training speedup on dense and MoE models for both supervised fine-tuning and reinforcement learning.

1 Introduction

Training Large Language Models (LLMs) often involves generating multiple output sequences from the same input: agentic workflows branch into parallel execution paths due to concurrent tool invocations, think-mode, and retokenization drift Luo et al. [2025]; self-consistency training samples K independent reasoning chains from one prompt Wang et al. [2022]; beam search distillation retains multiple candidate outputs; and multi-rollout SFT/RL collects several trajectories per query. In all these settings, the outputs share a common prefix—at minimum the prompt, and often substantial intermediate segments—so the tokens naturally form a **tree-structured trajectory** rather than a collection of independent linear sequences. Tree Training is designed to eliminate the redundant computation that arises in *any* such tree-structured training scenario.

The agentic setting provides our primary and most complex example. A single task frequently branches into multiple parallel paths driven by tree-structured workflow designs (e.g., tree-structured planning Yao et al. [2023], Zhang et al. [2024] and sub-agents Team et al. [2026]), concurrent tool invocations Kim et al. [2024], Abdelaziz et al. [2024], and context management Xu et al. [2025], Packer et al. [2024], Zhong et al. [2023]. These mechanisms imply that the context fed into turn $t{+}1$ is often *not* exactly the concatenation of the tokens produced in previous turns; therefore, the collected tokens form a tree-structured token trajectory with shared prefixes across branches. Existing training pipelines linearize such trajectories and treat each branch independently, leading to significant redundant forward/backward computation. Prefix caching Kwon et al. [2023], Pope et al. [2022] addresses this at inference time but does not extend to training.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

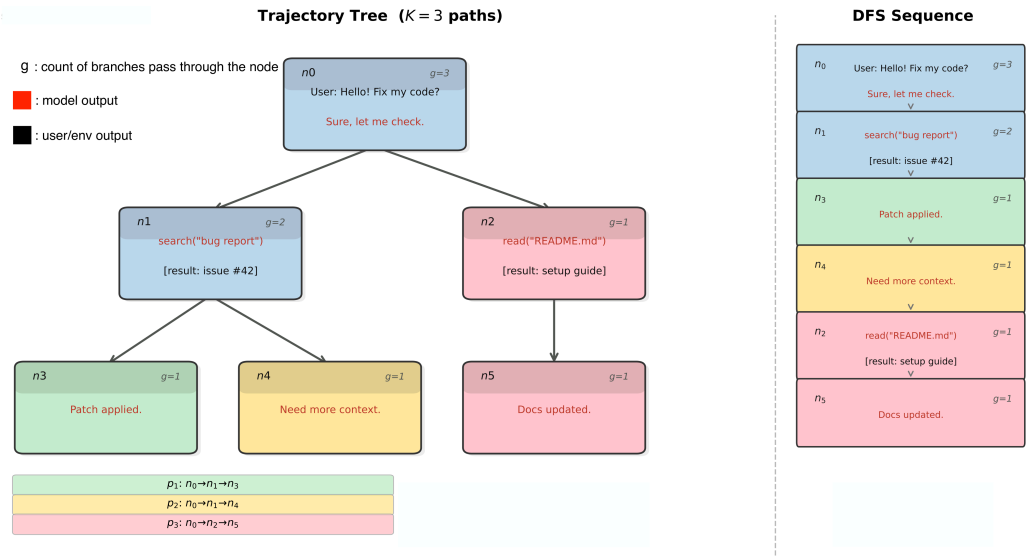

Figure 1: **Trajectory tree and DFS serialized sequence.** Left: a trajectory tree with $K = 3$ root-to-leaf paths. Each node n holds a token segment T_n; g_n is the number of paths through n (blue nodes are shared prefixes with $g_n > 1$). Node text in red is model output; black is user/environment input. Right: the DFS sequence places all nodes in depth-first order so that every token is computed exactly once, with no redundancy.

To address the above challenges, we note that by a simple reordering of summation (Eq. 4), training with shared prefixes *that participate in gradients* is equivalent to a per-token weighted loss. Prior prefix-reuse methods Liu et al. [2025], Wang and Hegde [2024], Goru et al. [2025] restrict themselves to gradient-free prefixes not because this algebra is unknown, but because achieving correct gradients for gradient-participating prefixes requires non-trivial model-layer adaptations: a tree-aware attention mask, depth-based position IDs, and State Space Model (SSM) Gu and Dao [2024], Dao and Gu [2024] state routing. We provide these adaptations, reducing the problem to computing each token's log-probability exactly once. When the full tree exceeds GPU memory, **Redundancy-Free Tree Partitioning** splits it into subtrees with differentiable partition boundaries. An alternative is to process each tree node individually via differentiable KV relay—this also achieves zero redundant computation but sacrifices GPU utilisation due to many small forward calls; we show the proposed DFS serialization approach is **3.88**$\times$ faster (Section 4.6). Our approach takes the best of both: DFS serialization within each partition ensures high compute density, while the differentiable boundaries between partitions enable trees larger than GPU memory with peak activation memory bounded by $\mathcal{O}(\text{max-path})$.

Together these form **Tree Training**, achieving up to **6.2**$\times$ end-to-end speedup in our experiments.

In summary, our contributions are:

(1) **DFS Tree Training.** We show that multi-branch training with gradient-participating shared prefixes reduces to per-token weighted loss, and provide the model-layer adaptations—tree attention mask, depth-based position IDs, and SSM state routing (the first treatment of branching in hybrid recurrent-attention models)—that make this hold in practice via DFS serialization.

(2) **Redundancy-Free Tree Partitioning.** For memory-constrained settings, we split the tree into DFS subtrees with differentiable partition boundaries, achieving zero redundant computation with peak memory $\mathcal{O}(\text{max-path})$. DFS within partitions is **3.88**$\times$ faster than per-node caching alternatives.

(3) We demonstrate up to **6.2**$\times$ end-to-end speedup on 30B/32B dense and MoE models without compromising gradient correctness or downstream task quality.

2 Related Work

Prefix caching is widely used to speed up autoregressive decoding by reusing identical prefixes in the forward pass Kwon et al. [2023], Pope et al. [2022], Liu et al. [2024], Cheng et al. [2024], Yao et al. [2025], Cheng et al. [2025], Qin et al. [2025]. For training objectives where the shared part is purely a prompt, batching strategies such as prefix grouping can also reuse the prompt prefix computation without affecting gradients Liu et al. [2025], Wang and Hegde [2024]; Tree Training strictly generalises this to gradient-participating prefixes and SSM hybrid architectures (Appendix E). In addition, Goru et al. Goru et al. [2025] deduplicate repeated reasoning tokens within a single conversation using a block-sparse mask, but assume the shared prefix does not contribute to the loss and no gradient flows through it—a strictly narrower problem that precludes direct comparison with our setting.

Prior works have also recognized that agentic/RL generation often forms tree-structured rollouts, and exploit shared prefixes primarily to improve *rollout* efficiency, e.g., tree-search style rollouts and prefix reuse Hou et al. [2025], Li et al. [2025], Ji et al. [2025]. However, the subsequent training stage is commonly executed on linearized samples: current training pipelines Luo et al. [2025], Zhang et al. [2025], Zhou et al. [2025] typically decompose a trajectory tree into separate linear segments and treat each segment independently, causing shared prefixes to be repeatedly recomputed in both forward and backward passes.

Concurrent tree-structured training systems. *AReaL-DTA* Zhang et al. [2026] belongs to the sequential-traversal family: a DFS stack materializes one root-to-leaf path at a time and runs forwards against the parent's cached KV (per segment, or per leaf suffix with chunked backward). This is the per-node execution model of Section 4.6, whose many small kernels under-utilize the GPU; it is also not strictly compute-once (up to N_{leaf} extra no-grad anchor forwards, plus local recomputation in chunked backward). AReaL-DTA argues against packed tree-mask execution because large trees must be split, reintroducing duplicated prefix computation; its advantage is measured against naive packing with forced splitting, full recomputation and a FlexAttention mask. Redundancy-Free Tree Partitioning (Section 3.3) removes exactly this failure mode while keeping dense packed execution. AReaL-DTA evaluates dense Qwen3 only, without parameter-gradient verification or hybrid SSM support. *Prompt Trees* Scaled Cognition [2025] shares our DFS packing, depth-based position IDs and block-sparse mask for in-memory trees, but establishes gradient equivalence only for leaf-only losses—the trivial case $g_t = 1$. For losses on shared tokens it observes their under-weighting and suggests an unspecified reweighting; Eq. (4) derives the exact correction g_t/K, making all-token training (Appendix D) exact rather than heuristic. It supports neither SSM hybrids nor memory-constrained trees. Each system thus lacks one of our two correctness pillars: an exact derivation for gradient-participating shared tokens, and parameter-level gradient verification (Section 4.4).

Sequence packing Krell et al. [2021] concatenates independent sequences with attention masks to prevent cross-sequence gradients. Tree Training generalises this to the harder case where sequences share a common prefix that participates in gradient updates—precisely the scenario sequence packing cannot address.

Recent hybrid LLM architectures interleave full-attention with State Space Model (SSM) layers; Gated Delta Net (GDN) Yang et al. [2025c] is adopted in models such as Qwen3.5 Team [2026]. Unlike attention, SSM layers maintain a recurrent hidden state requiring dedicated handling in tree-structured computation.

3 Tree Training

3.1 Derivation

A **trajectory tree** is a rooted tree in which each node n holds a token segment T_n. Each root-to-leaf path spells out a complete trajectory by concatenating the token segments of its nodes in order. Let $\mathcal{T}$ denote the set of all nodes and $\mathcal{P} = \{p_1, \ldots, p_K\}$ the set of K root-to-leaf paths (one per leaf). An example with $K = 3$ is shown in Figure 1. **Goal.** Process all tokens in the tree with each token computed *exactly once* (no redundant computation across shared prefixes) in a single forward and backward pass, and produce gradients that are mathematically equivalent to the baseline of running all K paths independently and averaging their losses.

The baseline sep-avg loss runs each of the K paths independently:

$$L_{\text{sep_avg}} = \frac{1}{K} \sum_{k=1}^{K} \sum_{t \in p_k} \ell_t(\theta) \tag{1}$$

where $\ell_t(\theta)$ is any per-token objective. For SFT, $\ell_t(\theta) = -\log p_\theta(y_t \mid x_{\le t})$; for RL with policy gradient, $\ell_t(\theta) = -A_t \cdot \log p_\theta(y_t \mid x_{\le t})$ where A_t is the per-token advantage.

Eq. (1) iterates over paths then over tokens in each path. We swap the order: iterate over every unique token t and count how many paths include it. Token t belongs to node n; let $g_n = |\{p \in \mathcal{P} : n \in p\}|$ be the number of paths passing through n, and $g_t \triangleq g_n$ for $t \in T_n$. Token t is therefore counted g_n times in the double sum:

$$\sum_{k=1}^{K} \sum_{t \in p_k} \ell_t(\theta) \;=\; \sum_{n \in \mathcal{T}} \sum_{t \in T_n} g_n \cdot \ell_t(\theta) \;=\; \sum_{t} g_t \cdot \ell_t(\theta) \tag{2}$$

Substituting (2) into (1):

$$L_{\text{sep_avg}} = \sum_{t} \frac{g_t}{K} \cdot \ell_t(\theta) \tag{3}$$

We therefore define the tree loss:

$$\boxed{L_{\text{tree}} = \sum_{t} \frac{g_t}{K} \cdot \ell_t(\theta)} \tag{4}$$

By linearity of differentiation:

$$\frac{\partial L_{\text{tree}}}{\partial \theta} = \sum_{t} \frac{g_t}{K} \cdot \frac{\partial \ell_t}{\partial \theta} = \frac{\partial L_{\text{sep_avg}}}{\partial \theta} \tag{5}$$

This derivation applies to any objective computable from per-token log-probabilities (including sequence-level rewards and per-token KL as in GRPO/PPO): each leaf path's loss uses its correctly-evaluated log-probabilities, and the aggregated gradient equals $\partial L_{\text{sep_avg}}/\partial\theta$. Objectives that jointly model distributions across multiple paths are outside scope. For PPO/GRPO: the clipping $\text{clip}(r_t, 1-\varepsilon, 1+\varepsilon)$ is per-token per-path and thus in scope; reference model log-probabilities are computed via a standard per-path forward on the frozen reference model, unaffected by this framework.

Problem reduction. The goal therefore reduces to: find a method that (1) computes each token *exactly once* with no redundant computation, and (2) ensures that $\ell_t(\theta)$ computed this way matches what it would be in a standalone per-branch forward pass—i.e., **forward equivalence**:

$$\ell_t^{\text{tree}}(\theta) = \ell_t^{p}(\theta) \quad \forall\, p \in \text{paths}(n_t) \tag{6}$$

Forward equivalence is a *constructive* design goal: we design the attention mask, position embeddings, and SSM state routing so that each token observes exactly the same context it would in an independent per-branch forward, and verify this empirically on Qwen3 (pure attention), Qwen3.5 (SSM hybrid with GDN layers), and Llama-3.2 Grattafiori et al. [2024] (pure transformer; Appendix C); see Appendix I. We achieve this via DFS traversal of the token prefix tree, which serializes all nodes in depth-first order so each token appears exactly once. DFS is chosen over BFS or topological-level batching because backtracking releases completed subtrees' activations immediately, bounding peak memory to $\mathcal{O}(\text{max-path tokens})$; BFS would require retaining all nodes at the same depth simultaneously.

$$X_{\text{base}}(i) \quad = \quad \text{Concat}\big[\text{token}(i),\, X_{\text{base}}(\text{child}(0,i)), \text{token}(i),\, X_{\text{base}}(\text{child}(1,i)), \quad \ldots\big] \tag{7}$$

with:

$$X_{\text{DFS}}(i) \quad = \quad \text{Concat}\big[\text{token}(i),\, X_{\text{DFS}}(\text{child}(0,i)), X_{\text{DFS}}(\text{child}(1,i)), \quad \ldots\big] \tag{8}$$

How to enforce (6) on the **DFS-serialized** sequence—i.e., ensuring each token's log-probability matches its per-branch value despite the reordering introduced by DFS traversal—is described as follows.

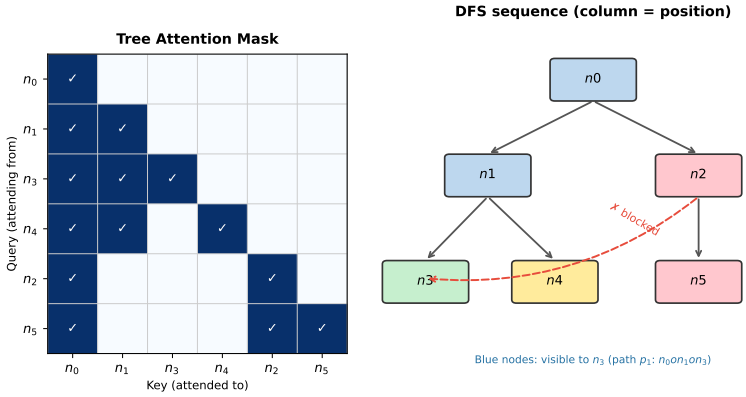

Figure 2: **Tree attention mask.** Left: the 6×6 attention matrix for the DFS sequence; a cell is filled iff the query token can attend to the key token. Tokens from different branches (e.g. n_3 and n_2) are mutually blocked. Right: the corresponding tree structure; the dashed red arrow shows a blocked cross-branch path.

3.2 Implementation

Attention Mask. We construct a tree attention mask such that token i attends to token j iff (1) $j \leq i$ and (2) the nodes of i and j lie on a common root-to-leaf path. This prevents cross-branch attention. The attention pattern is illustrated in Figure 2.

Position Embedding. Each token must receive the same position ID as it would in its standalone per-path sequence. A token at offset j within node n is assigned:

$$\text{pos}(t) = \sum_{n' \in \text{anc}(n) \setminus \{n\}} |T_{n'}| + j \tag{9}$$

where $\text{anc}(n)$ denotes the ancestors of n. Sibling nodes at the same depth share the same position range, matching the per-path baseline and ensuring RoPE Su et al. [2023] produces identical results.

SSM Layers (GDN). For hybrid models that interleave full-attention with Gated Delta Net (GDN) Yang et al. [2025c] SSM layers, the attention mask alone is insufficient: GDN layers maintain a recurrent hidden state passed sequentially between chunk boundaries, so a naïve DFS layout would carry the state from one branch into the next sibling branch. We apply two fixes. We treat each tree node's token segment as one *chunk*—the unit of SSM state transfer.

(i) **State routing**: each chunk c reads its initial SSM state from its *parent* chunk $\pi(c)$ rather than the preceding chunk in DFS order:

$$\mathbf{h}_c^{(0)} = \mathbf{h}_{\pi(c)}^{(\text{end})}, \qquad \mathbf{h}_{-1}^{(\text{end})} \triangleq \mathbf{0} \tag{10}$$

where $\mathbf{h}_c^{(0)}$ is the initial recurrent state of chunk c, $\mathbf{h}_c^{(\text{end})}$ is its final state after processing all tokens in the chunk, and the convention $\mathbf{h}_{-1}^{(\text{end})} \triangleq \mathbf{0}$ initialises root chunks (those with no parent) to the zero state. DFS order guarantees $\pi(c) < c$, so parent states are always ready. Sibling chunks share the same parent state tensor, so their gradients accumulate there automatically during the backward pass; because the backward is executed sequentially (one leaf at a time with `retain_graph`), there is no concurrency and gradient accumulation is exact. Figure 3 illustrates the difference between sequential and tree state routing.

(ii) **Causal convolution**: SSM layers include a causal conv1d with kernel size K_{conv}, which requires the preceding $K_{\text{conv}}-1$ tokens as left context. In the DFS sequence, the tokens immediately before a child chunk belong to a sibling branch rather than the ancestor path, so using them as context is incorrect. Instead, each chunk c uses $\mathbf{ctx}_{\pi(c)}$—the last $K_{\text{conv}}-1$ output tokens of its parent chunk $\pi(c)$, saved after processing the parent—as its left context:

$$\mathbf{y}_c = \text{Conv}\big([\,\mathbf{ctx}_{\pi(c)}\,;\,\mathbf{x}_c\,]\big)_{[K_{\text{conv}}-1:]} \tag{11}$$

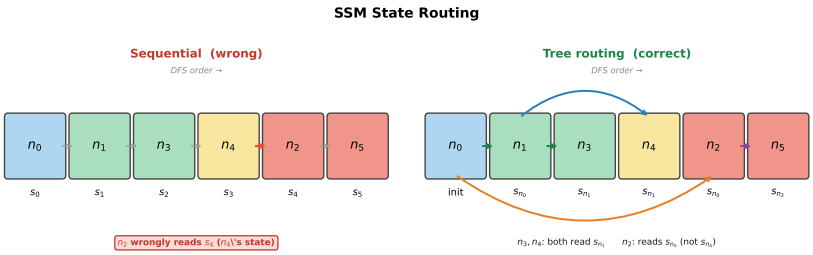

Figure 3: **SSM state routing.** Sequential flow (left) feeds n_4's state into n_2—wrong, since n_2 is a sibling of n_1, not a descendant of n_4. Tree routing (right) routes each chunk to its parent's state: both n_3 and n_4 read s_{n_1}, while n_2 correctly reads s_{n_0}.

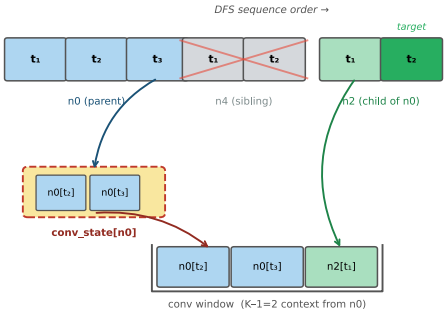

Figure 4: **SSM causal convolution** ($K_{\text{conv}} = 3$). When processing n_2 (child of n_0), the correct conv window is formed by prepending `conv_state`$[n_0]$—the stored last $K_{\text{conv}} - 1$ effective tokens of the parent—as left context, bypassing the DFS-adjacent sibling tokens (n_4) that occupy the preceding positions.

This replicates the context each token would see in an independent per-branch forward pass. Figure 4 illustrates this for $K_{\text{conv}} = 3$.

Loss Weights. For each token t in the DFS-serialized sequence, its per-token loss $\ell_t(\theta)$ is multiplied by a weight λ_t before reduction. The specific value of λ_t depends on the training objective: for the sep-avg baseline with equal path weights this gives $\lambda_t = g_t/K$ as derived in Eq. (4); in general λ_t can be any value determined by the path weighting scheme (see Section 3.1). This is applied as a single element-wise multiplication on the per-token loss tensor—requiring no modification to the backward pass and negligible overhead. Full implementation details appear in Appendix I.

3.3 Redundancy-Free Tree Partitioning

When N_{tree} exceeds GPU memory capacity, the tree must be split into subtrees, each processed as a single DFS forward pass, with partition boundaries made differentiable to eliminate boundary recomputation and achieve zero redundant computation with peak memory bounded by a single root-to-leaf path.

Two design questions arise: (1) how to partition optimally while keeping peak activation memory bounded; and (2) how to avoid redundant recomputation of shared ancestor tokens at partition boundaries. (e.g. Figure 5).

Partitioning. Partitions must follow tree node boundaries: each partition is a connected subtree, and every cut occurs at a node boundary. The reason is peak activation memory. The backward pass chains gradients bottom-up through the partition dependency graph; to bound peak memory at $\mathcal{O}(\text{max-path tokens})$, this graph must itself be a tree—meaning each partition has exactly one parent partition. This holds if and only if cuts are at node boundaries. Arbitrary cuts would place tokens from independent subtrees into the same partition; its backward must then simultaneously retain

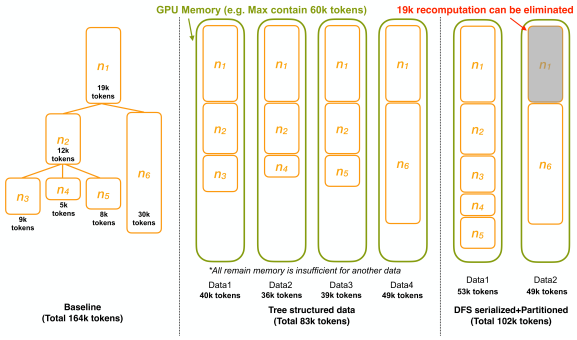

Figure 5: Memory-constrained training on a tree with 83k unique tokens (GPU limit $C = 60$k). Baseline flattening: 164k tokens. Standard Tree Partitioning (no differentiable boundaries): 102k tokens due to boundary recomputation. With differentiable partition boundaries, boundary recomputation is eliminated entirely: 83k tokens, equal to the tree's unique token count.

the computation graphs of *multiple* parent partitions (and transitively their ancestors), potentially keeping the graphs of all ancestor subtrees of all branches in memory at once.

The optimization goal is therefore to minimise $N_{\text{partitions}}$—equivalently, maximise token utilisation of each partition—subject to each partition containing at most C tokens. This is a standard bin packing problem on tree subgraphs, solvable with OR-Tools Perron and Furnon [2024]; partitioning is computed offline once per collected tree and adds negligible overhead (milliseconds) relative to the forward/backward pass.

Differentiable partition boundaries. The second problem—avoiding redundant recomputation of shared prefixes across subtrees—is solved by making partition boundaries differentiable. At each cut node n_c, the parent partition's accumulated KV tensors (and SSM states for hybrid models) are detached and re-exposed as differentiable leaf variables; the child partition attends to these cached ancestor key-value pairs without re-executing any ancestor token. After the child partition's backward, gradients in the leaf tensors are passed back into the parent partition's retained computation graph via an explicit backward pass through the parent—the same gradient-relay mechanism used in pipeline-parallel training—so each token is computed exactly once and total computation equals N_{tree} (Figure 5: 102k $\rightarrow$ 83k). For SSM hybrid models, the gateway additionally carries the recurrent hidden state and causal-conv context at the cut node, chained back identically. Details (ancestor-aware attention mask, depth-based position offset, float32 accumulation for multi-child branching, SSM state injection) are listed in Appendix J.

DFS serialization versus per-node caching. The differentiable boundary mechanism above can be applied at *every* tree node boundary—not just partition boundaries—yielding *per-node caching*: the model is called N_{nodes} times, each processing one node's own token segment, with the accumulated KV relayed from the parent node. Both approaches achieve zero redundant computation; DFS serialization is preferable for two reasons. First, DFS processes tokens in $\mathcal{O}(N_{\text{partitions}})$ calls instead of N_{nodes}, directly reducing CPU/kernel-launch overhead. Second, each DFS call covers far more tokens, making operations such as linear projections (GEMMs) significantly denser: in per-node caching, each call reloads the weight matrix with only the current node's tokens to amortise it, while DFS amortises the same weight load over the entire partition. For SSM hybrid models, in per-node caching the recurrent state is written to and reloaded from HBM at every node boundary; DFS keeps it on-chip throughout the partition, avoiding the dominant memory bottleneck of SSM layers entirely Gupta et al. [2026]. Experimentally, DFS packing is **3.88**$\times$ faster than per-node caching on identical workloads (Section 4.6).

4 Experiments

4.1 Metrics

Potential Overlap Ratio (POR) quantifies the theoretical upper bound of computation reuse by measuring the redundancy inherent in shared prefixes. We define POR as the ratio of overlapping tokens to the total token count of the flattened dataset:

$$\text{POR} = 1 - \frac{N_{\text{tree}}}{N_{X_{base}(root)}} \tag{12}$$

Here, N_{tree} denotes the token count in the prefix tree, while $N_{X_{base}(root)}$ represents the token counts of the serialized tree produced by the baseline method described in Equation (1). Using the example from Figure 5, the POR is calculated as $1 - \frac{83\text{k}}{164\text{k}} \approx \mathbf{49}.\mathbf{4}\%$, indicating that nearly half of the baseline computation is redundant. Since Redundancy-Free Tree Partitioning computes each token exactly once, the theoretical speedup upper bound is $1/(1 - \text{POR})$ (a FLOP-count bound; small wall-time gaps arise from gateway overhead and partition-size effects).

4.2 Evaluation Setups

We evaluate the end-to-end training efficiency by comparing the total time required to process equivalent data between Tree Training and the baseline described in Section 3. This baseline aligns with the current standard practice for training on branched trajectories, as most existing training frameworks only support taking batch of **list** (tensor) as training inputs. Although our experiments focus on agentic trajectories, the same framework and speedup bounds ($1/(1 - \text{POR})$) apply to any tree-structured training data including self-consistency, beam search distillation, and multi-rollout SFT/RL, with POR determined by the fraction of tokens in shared prefixes. Note that training-side prefix grouping Liu et al. [2025] is inapplicable here: it reuses a shared prompt that does not contribute to gradients, whereas our shared prefixes are intermediate trajectory segments that aggregate gradient contributions from multiple branches.

All experiments were conducted on clusters of 64 NVIDIA Hopper GPUs using Megatron-Core Shoeybi et al. [2020] for distributed training. Tensor, expert, data, and pipeline parallelism are *orthogonal* to Tree Training: the tree attention mask, DFS position IDs, and SSM routing operate within each device's model shard without additional inter-rank coordination. Sequence Packing Krell et al. [2021] is used in our baseline setting, representing the standard batching strategy used in large-scale LLM fine-tuning and RL training. We release the reference implementation and the gradient-verification harness at `https://github.com/Jinghui-Wang/tree-training`.

4.3 Speedup and Correctness on real-world agentic trajectories

To assess real-world applicability, we collect agentic RL rollouts from multi-turn reinforcement learning tasks. Figure 6 presents representative trajectory trees alongside token-overlap distributions, covering POR from 28.0% to 88.7%. These trajectories are collected from software engineering tasks (SWE-smith Yang et al. [2025b]), executed via agentic scaffolds Claude Code Anthropic [2025]. Most branching behaviors of the left two trees are caused by concurrent tool execution and retokenization drift Luo et al. [2025], while the right tree is caused by the think mode. Context compaction does not break the tree: it adds a new subtree at the end of the system prompt and changes only POR, never correctness (Appendix F). In practice, POR exceeds 80% whenever think-mode is enabled and intermediate reasoning tokens are not fully preserved across turns—the dominant pattern in modern agentic systems and the scenario illustrated by the high-overlap tree (POR=88.7%) in our SWE-smith data.

(1) Speedup (Figure 7, Top): The theoretical speedup upper bound $\frac{1}{(1-\text{POR})}$ for this dataset is **6.5**×. We achieved average realized speedups of **6.3**× (32B Dense) and **6.2**× (30B MoE), capturing over **95%** of the theoretical potential. The step-wise fluctuation (2×10×) faithfully reflects the varying intrinsic POR of dynamic tree samples. The additional tensors introduced by Tree Training total only **1.2 MB** on Qwen3.5-35B-A3B, confirming negligible memory overhead (see Section 4.6).

(2) No cross-step drift (Figure 7, Bottom): This curve does *not* test per-step gradient equivalence—that is established at the parameter level in Section 4.4. Its purpose is to check that small per-step

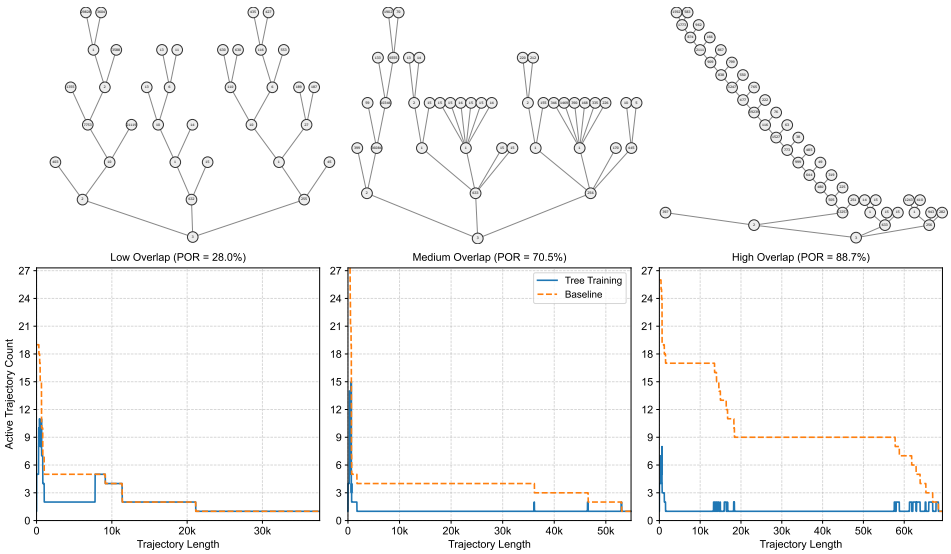

Figure 6: **Real agentic trajectory trees and their overlap characteristics.** The upper row shows representative trees from multi-turn agentic RL rollouts with **Low**, **Medium**, and **High Overlap**. The lower row plots *active trajectory counts* over length; the area ratio reflects the theoretical token reuse ratio.

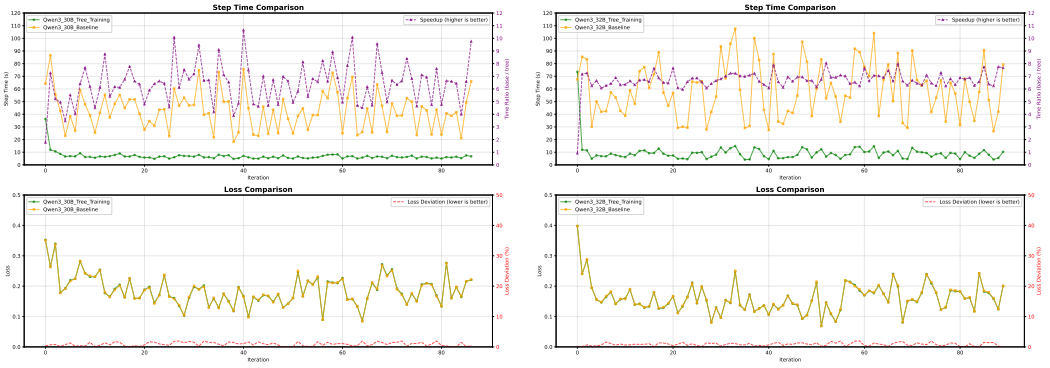

Figure 7: **Training speedup and loss comparison using real world rollout data with think mode turned on.** Results for the MoE model Qwen3-30B Yang et al. [2025a] are shown on the left, and for the dense model Qwen3-32B on the right. For each model, the top plot illustrates the training speedup achieved by Tree Training, while the bottom plot displays the average relative error in loss.

numerical differences do not *accumulate* over training. Starting from the same base checkpoint, Tree Training and the baseline track each other from step 0; in bfloat16 training the loss noise floor is ≈ 0.005, and the measured per-step deviation (red dashed) stays below 1% with no upward trend. Training on all branch tokens—enabled by Tree Training's memory efficiency—further yields a **+37%** quality gain on Terminal Bench 2.0 compared to single-branch training (Appendix D).

4.4 Parameter-Level Gradient Equivalence

Equal scalar losses do not imply equal parameter gradients, so we compare gradients directly. For every parameter we capture the gradient via float32 hooks under deterministic kernels (deterministic algorithms, fixed cuBLAS workspace, TF32 off) and report the maximum relative difference against the independent per-branch baseline (A), aggregated per layer group. Because any re-layout of the same computation changes GEMM reduction order, we calibrate against two baseline-vs-itself noise floors: the *reorder floor* (A with branch order reversed; pure accumulation-order noise) and the

Table 1: **Layer-wise parameter-gradient equivalence (FP32).** Max relative gradient difference vs. independent per-branch forwards, per layer group. Hybrid: hybrid GDN model with 18 of 24 layers GDN; attention-only: Qwen3-4B. Tree Training and all partitioned variants sit at the magnitude of the baseline's own shape floor, and no group degrades as partitioning becomes more aggressive.

Model	Comparison	Embed	Attn	GDN	MLP	Norms	Overall
Hybrid GDN	Reorder floor (A vs. A′)	6.8e-8	1.4e-7	2.0e-7	1.8e-7	1.2e-7	2.0e-7
	Shape floor (A vs. batched A)	3.0e-6	2.5e-5	2.2e-5	1.7e-5	2.3e-5	2.5e-5
	Tree Training, no partition	1.3e-5	6.7e-5	8.1e-5	6.4e-5	8.3e-5	**8.3e-5**
	Partitioned, 4 subtrees	3.7e-6	2.4e-5	2.2e-5	1.8e-5	3.2e-5	**3.2e-5**
	Partitioned, 7 subtrees	5.6e-6	3.1e-5	1.7e-5	1.6e-5	2.1e-5	**3.1e-5**
Attention-only	Reorder floor (A vs. A′)	6.4e-8	1.7e-7	–	1.6e-7	1.4e-7	1.7e-7
	Shape floor (A vs. batched A)	1.9e-5	4.8e-5	–	3.6e-5	5.5e-5	5.5e-5
	Tree Training, no partition	3.2e-6	6.3e-5	–	1.6e-5	9.2e-5	**9.2e-5**
	Partitioned, 4 subtrees	2.4e-6	3.1e-5	–	8.7e-6	3.3e-5	**3.3e-5**
	Partitioned, 7 subtrees	6.6e-6	1.7e-5	–	1.3e-5	2.8e-5	**2.8e-5**

shape floor (the identical per-branch computation laid out as one padded batch; same math, different GEMM shapes), which is the fair floor for any packed method.

Table 1 shows that (i) in FP32, Tree Training and every partitioned variant match per-branch gradients to $\leq 9.2 \times 10^{-5}$ relative error, the magnitude of the baseline's own shape floor, with no systematic offset; and (ii) no layer group, in particular neither GDN nor attention, degrades as partitioning becomes more aggressive ($4 \rightarrow 7$ subtrees), which rules out a systematic loss of small child-partition gradients at cut nodes (e.g., flush-to-zero when relayed gradients are re-rounded to BF16). (iii) In BF16, overall deviations (hybrid / attention-only: Tree Training 1.4e-1 / 5.1e-1, partitioned 1.4–1.5e-1 / 2.5–3.3e-1) lie in the band of the baseline's own BF16 shape floor (6.5e-2 / 4.2e-1), identifying bfloat16 non-associativity, not the method, as their source. Appendix C repeats the check on Llama-3.2.

4.5 Speedup of datasets with varying POR

On synthetic datasets with POR from 20% to 92% (fixed leaf count and total tokens), the speedup grows monotonically with POR, up to $8.7\times$ when full trees fit in memory (Appendix A). A controlled sweep on a plain PyTorch stack (Appendix G) shows that speedup tracks $1/(1-\text{POR})$ across tree shapes, is insensitive to batch size beyond POR and to node granularity ($< 1\%$), and degrades smoothly, without a cliff, only when a tree is forced into many small partitions.

4.6 DFS Packing vs. Per-Node Caching, and Memory Footprint

On Qwen3.5-35B-A3B with 256 trees per batch (13 nodes/tree on average), DFS packing achieves a median step time of **1448 ms** versus **5626 ms** for per-node caching—a **3.88**$\times$ speedup, winning on all 47 steps with a $3\times$ smaller step-time standard deviation (440 vs. 1358 ms; Appendix B). The only additional memory of Tree Training is the gateway tensors at partition boundaries: **1.2 MB** per tree on Qwen3.5-35B-A3B, under 0.002% of the $\approx$64 GB minimum activation memory.

5 Conclusion

Tree Training eliminates redundant computation in training on tree-structured trajectories by processing each shared prefix exactly once, while preserving exact gradient equivalence to the per-branch baseline—including when the prefix participates in gradient updates. Appendix H discusses extensions to MLA, sparse/compressed, sliding-window and other recurrent attention; looped transformers Fan et al. [2025] and prefix-aware scheduling across devices and training steps are left for future work.

Acknowledgments and Disclosure of Funding

Computational resources for this work were provided by Kuaishou Technology. The authors declare no competing interests.

References

I. Abdelaziz, K. Basu, M. Agarwal, S. Kumaravel, M. Stallone, R. Panda, Y. Rizk, G. P. S. Bhargav, M. Crouse, C. Gunasekara, S. Ikbal, S. Joshi, H. Karanam, V. Kumar, A. Munawar, S. Neelam, D. Raghu, U. Sharma, A. M. Soria, D. Sreedhar, P. Venkateswaran, M. Unuvar, D. D. Cox, S. Roukos, L. A. Lastras, and P. Kapanipathi. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. In F. Dernoncourt, D. Preoţiuc-Pietro, and A. Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1131–1139, Miami, Florida, US, Nov. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.85. URL https://aclanthology.org/2024.emnlp-industry.85/.

Anthropic. Claude code. https://claude.ai/code, 2025.

Y. Cheng, K. Du, J. Yao, and J. Jiang. Do large language models need a content delivery network? *arXiv preprint arXiv:2409.13761*, 2024.

Y. Cheng, Y. Liu, J. Yao, Y. An, X. Chen, S. Feng, Y. Huang, S. Shen, K. Du, and J. Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

T. Dao and A. Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

Y. Fan, Y. Du, K. Ramchandran, and K. Lee. Looped transformers for length generalization, 2025. URL https://arxiv.org/abs/2409.15647.

R. Goru, S. Mehta, and P. Jain. One-pass to reason: Token duplication and block-sparse mask for efficient fine-tuning on multi-turn reasoning, 2025. URL https://arxiv.org/abs/2504.18246.

A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez,

V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma. The llama 3 herd of models, 2024. URL `https://arxiv.org/abs/2407.21783`.

A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL `https://arxiv.org/abs/2312.00752`.

N. Gupta, P. Wang, R. Kannan, and V. K. Prasanna. A persistent-state dataflow accelerator for memory-bound linear attention decode on FPGA, 2026. URL `https://arxiv.org/abs/2603.05931`.

Z. Hou, Z. Hu, Y. Li, R. Lu, J. Tang, and Y. Dong. Treerl: Llm reinforcement learning with on-policy tree search, 2025. URL `https://arxiv.org/abs/2506.11902`.

Y. Ji, Z. Ma, Y. Wang, G. Chen, X. Chu, and L. Wu. Tree search for llm agent reinforcement learning, 2025. URL `https://arxiv.org/abs/2509.21240`.

S. Kim, S. Moon, R. Tabrizi, N. Lee, M. W. Mahoney, K. Keutzer, and A. Gholami. An llm compiler for parallel function calling, 2024. URL `https://arxiv.org/abs/2312.04511`.

M. M. Krell, M. Kosec, S. P. Perez, and A. Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *arXiv preprint arXiv:2107.02027*, 2021.

W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Y. Li, Q. Gu, Z. Wen, Z. Li, T. Xing, S. Guo, T. Zheng, X. Zhou, X. Qu, W. Zhou, Z. Zhang, W. Shen, Q. Liu, C. Lin, J. Yang, G. Zhang, and W. Huang. Treepo: Bridging the gap of policy optimization and efficacy and inference efficiency with heuristic tree-based modeling, 2025. URL https://arxiv.org/abs/2508.17445.

Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56, 2024.

Z. Liu, T. Yue, Y. Tang, L. Guo, J. Cai, Q. Liu, X. Chen, and J. Liu. Prefix grouper: Efficient grpo training through shared-prefix forward, 2025. URL https://arxiv.org/abs/2506.05433.

X. Luo, Y. Zhang, Z. He, Z. Wang, S. Zhao, D. Li, L. K. Qiu, and Y. Yang. Agent lightning: Train any ai agents with reinforcement learning, 2025. URL https://arxiv.org/abs/2508.03680.

M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. M. Rytting, R. Marten, Y. Wang, A. Dimakis, A. Konwinski, and L. Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

L. Perron and V. Furnon. Or-tools. https://developers.google.com/optimization/, 2024.

R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference, 2022. URL https://arxiv.org/abs/2211.05102.

R. Qin, Z. Li, W. He, M. Zhang, Y. Wu, W. Zheng, and X. Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2025. URL https://arxiv.org/abs/2407.00079.

Scaled Cognition. Prompt trees: Training-time prefix caching. https://www.scaledcognition.com/blog/prompt-trees, 2025.

M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

K. Team, T. Bai, Y. Bai, Y. Bao, S. H. Cai, Y. Cao, Y. Charles, H. S. Che, C. Chen, G. Chen, H. Chen, J. Chen, J. Chen, J. Chen, J. Chen, K. Chen, L. Chen, R. Chen, X. Chen, Y. Chen, Y. Chen, Y. Chen, Y. Chen, Y. Chen, Y. Chen, Y. Chen, Z. Chen, Z. Chen, D. Cheng, M. Chu, J. Cui, J. Deng, M. Diao, H. Ding, M. Dong, M. Dong, Y. Dong, Y. Dong, A. Du, C. Du, D. Du, L. Du, Y. Du, Y. Fan, S. Fang, Q. Feng, Y. Feng, G. Fu, K. Fu, H. Gao, T. Gao, Y. Ge, S. Geng, C. Gong, X. Gong, Z. Gongque, Q. Gu, X. Gu, Y. Gu, L. Guan, Y. Guo, X. Hao, W. He, W. He, Y. He, C. Hong, H. Hu, J. Hu, Y. Hu, Z. Hu, K. Huang, R. Huang, W. Huang, Z. Huang, T. Jiang, Z. Jiang, X. Jin, Y. Jing, G. Lai, A. Li, C. Li, C. Li, F. Li, G. Li, G. Li, H. Li, H. Li, J. Li, J. Li, J. Li, L. Li, M. Li, W. Li, W. Li, X. Li, X. Li, Y. Li, Y. Li, Y. Li, Y. Li, Z. Li, Z. Li, W. Liao, J. Lin, X. Lin, Z. Lin, Z. Lin, C. Liu, C. Liu, H. Liu, L. Liu, S. Liu, S. Liu, S. Liu,

T. Liu, T. Liu, W. Liu, X. Liu, Y. Liu, Y. Liu, Y. Liu, Y. Liu, Y. Liu, Z. Liu, Z. Liu, E. Lu, H. Lu, Z. Lu, J. Luo, T. Luo, Y. Luo, L. Ma, Y. Ma, S. Mao, Y. Mei, X. Men, F. Meng, Z. Meng, Y. Miao, M. Ni, K. Ouyang, S. Pan, B. Pang, Y. Qian, R. Qin, Z. Qin, J. Qiu, B. Qu, Z. Shang, Y. Shao, T. Shen, Z. Shen, J. Shi, L. Shi, S. Shi, F. Song, P. Song, T. Song, X. Song, H. Su, J. Su, Z. Su, L. Sui, J. Sun, J. Sun, T. Sun, F. Sung, Y. Tai, C. Tang, H. Tang, X. Tang, Z. Tang, J. Tao, S. Teng, C. Tian, P. Tian, A. Wang, B. Wang, C. Wang, C. Wang, C. Wang, D. Wang, D. Wang, D. Wang, F. Wang, H. Wang, H. Wang, H. Wang, H. Wang, H. Wang, J. Wang, J. Wang, J. Wang, K. Wang, L. Wang, Q. Wang, S. Wang, S. Wang, S. Wang, W. Wang, X. Wang, X. Wang, Y. Wang, Y. Wang, Y. Wang, Y. Wang, Y. Wang, Y. Wang, Z. Wang, Z. Wang, Z. Wang, Z. Wang, Z. Wang, Z. Wang, C. Wei, M. Wei, C. Wen, Z. Wen, C. Wu, H. Wu, J. Wu, R. Wu, W. Wu, Y. Wu, Y. Wu, Y. Wu, Z. Wu, C. Xiao, J. Xie, X. Xie, Y. Xie, Y. Xin, B. Xing, B. Xu, J. Xu, J. Xu, J. Xu, L. H. Xu, L. Xu, S. Xu, W. Xu, X. Xu, X. Xu, Y. Xu, Y. Xu, Y. Xu, Z. Xu, Z. Xu, J. Yan, Y. Yan, G. Yang, H. Yang, J. Yang, K. Yang, N. Yang, R. Yang, X. Yang, X. Yang, Y. Yang, Y. Yang, Y. Yang, Z. Yang, Z. Yang, Z. Yang, H. Yao, D. Ye, W. Ye, Z. Ye, B. Yin, C. Yu, L. Yu, T. Yu, T. Yu, E. Yuan, M. Yuan, X. Yuan, Y. Yue, W. Zeng, D. Zha, H. Zhan, D. Zhang, H. Zhang, J. Zhang, P. Zhang, Q. Zhang, R. Zhang, X. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, Z. Zhang, C. Zhao, F. Zhao, J. Zhao, S. Zhao, X. Zhao, Y. Zhao, Z. Zhao, H. Zheng, R. Zheng, S. Zheng, T. Zheng, J. Zhong, L. Zhong, W. Zhong, M. Zhou, R. Zhou, X. Zhou, Z. Zhou, J. Zhu, L. Zhu, X. Zhu, Y. Zhu, Z. Zhu, J. Zhuang, W. Zhuang, Y. Zou, and X. Zu. Kimi k2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Q. Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

F. Wang and S. Hegde. Accelerating direct preference optimization with prefix sharing, 2024. URL https://arxiv.org/abs/2410.20305.

G. Wang, J. Zeng, X. Xiao, S. Wu, J. Yang, L. Zheng, Z. Chen, J. Bian, D. Yu, and H. Wang. Flashmask: Efficient and rich mask extension of flashattention, 2025. URL https://arxiv.org/abs/2410.01359.

X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

W. Xu, K. Mei, H. Gao, J. Tan, Z. Liang, and Y. Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

J. Yang, K. Lieret, C. E. Jimenez, A. Wettig, K. Khandpur, Y. Zhang, B. Hui, O. Press, L. Schmidt, and D. Yang. Swe-smith: Scaling data for software engineering agents, 2025b. URL https://arxiv.org/abs/2504.21798.

S. Yang, J. Kautz, and A. Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025c. URL https://arxiv.org/abs/2412.06464.

J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 94–109, 2025. doi: 10.1145/3689031.3696098.

S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

D. Zhang, S. Zhoubian, Z. Hu, Y. Yue, Y. Dong, and J. Tang. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37: 64735–64772, 2024.

H. Zhang, X. Liu, B. Lv, X. Sun, B. Jing, I. L. Iong, Z. Hou, Z. Qi, H. Lai, Y. Xu, R. Lu, H. Wang, J. Tang, and Y. Dong. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework, 2025. URL https://arxiv.org/abs/2510.04206.

J. Zhang, Y. Yang, R. Yan, Z. Mei, L. Zhang, D. Li, W. Fu, J. Gao, S. Xu, Y. Wu, and B. Yuan. AReaL-DTA: Dynamic tree attention for efficient reinforcement learning of large language models. *arXiv preprint arXiv:2602.00482*, 2026.

W. Zhong, L. Guo, Q. Gao, H. Ye, and Y. Wang. Memorybank: Enhancing large language models with long-term memory, 2023. URL https://arxiv.org/abs/2305.10250.

Y. Zhou, S. Jiang, Y. Tian, J. Weston, S. Levine, S. Sukhbaatar, and X. Li. Sweet-rl: Training multi-turn llm agents on collaborative reasoning tasks, 2025. URL https://arxiv.org/abs/2503.15478.

A Speedup across Datasets with Varying POR

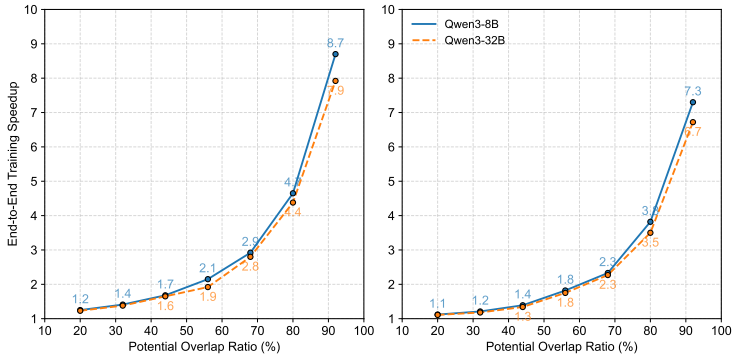

Figure 8: **End-to-end training speedup of Tree Training across datasets with varying POR.** Each subfigure reports the relative reduction in total training time of Tree Training compared to the baseline. (a) synthetic datasets where the full tree fits in GPU memory, (b) synthetic datasets requiring tree partitioning under memory constraints.

B DFS Packing vs. Per-Node Processing: Timing Benchmark

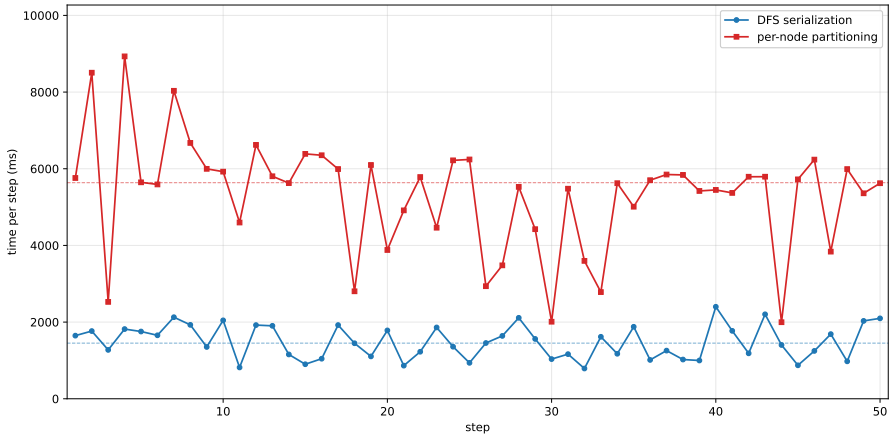

Figure 9: **Per-step wall time: DFS packing vs. per-node caching (Qwen3.5-35B-A3B, 256 trees/batch, avg. 13 nodes/tree, 47 steps).** Both methods receive identical inputs. Dashed horizontal lines mark the medians. DFS packing (blue) achieves a median of 1448 ms; per-node caching (red) achieves 5626 ms—a **3.88**× speedup. Per-node processing also exhibits 3× larger step-time variance (std 1358 ms vs. 440 ms), making it less predictable. DFS packing wins on all 47 steps.

C Llama Architecture Verification

To confirm that Tree Training is not specific to the Qwen model family, we run the same gradient check (methods A/B/C from Appendix I) on a standard Llama-style dense-attention transformer (Llama-3.2-3B, no SSM layers).

Configuration	max rel diff	Conclusion
no partition (max_tokens=1000)	9.5×10^{-5}	below noise floor
moderate cut (max_tokens=20)	1.6×10^{-4}	below noise floor
aggressive cut (max_tokens=10)	1.6×10^{-4}	below noise floor

All deviations are well below the bfloat16 training noise floor (≈ 0.005), confirming gradient correctness on standard dense-attention architectures.

D Training on All Branch Tokens Improves Model Quality

We provide a downstream evaluation to assess the quality impact of training on all branch tokens versus a single branch. Training on all tokens of trajectory trees leads to a clear performance gain—particularly for thinking models where branches share substantial prefixes.

On Terminal Bench 2.0 Merrill et al. [2026], we use a cold-started Qwen3-32B Yang et al. [2025a] as the base checkpoint for RL training with think-mode enabled. Training on the full tree achieves **28.8** (avg@4) versus **20.9** for the single-branch baseline—a relative improvement of $+37\%$—with identical environment interactions in both conditions. This confirms that the gradient equivalence provided by Tree Training translates into genuine task-level quality gains when all branch tokens can be leveraged.

E Relationship to Prefix Grouper

Liu et al. Liu et al. [2025] (Prefix Grouper) accelerates GRPO by grouping samples that share a common prompt and running a single forward pass over the shared prefix. We clarify the relationship to Tree Training along two dimensions.

Dense-attention, single-turn setting. When training data consists of K independent completions from the same prompt (e.g., standard single-turn GRPO on a dense-attention model), the trajectory tree degenerates to a single shared root (the prompt) with K leaf branches. In this case, Tree Training and Prefix Grouper are not merely equivalent in effect—they produce *identical implementations*. The DFS sequence for a one-level tree is simply the prompt followed by all K completions concatenated in order; the tree attention mask reduces to exactly the block-diagonal mask used by Prefix Grouper (prompt visible to all, completions causally masked and mutually invisible). Prefix Grouper is therefore a special case of DFS serialization restricted to single-level trees with a gradient-free root.

SSM hybrid architectures. For models that interleave attention with State Space Model (SSM) layers (e.g., Qwen3.5 with Gated Delta Net), Prefix Grouper has no mechanism to handle the SSM recurrent state at branch points: the recurrent state of the shared prefix must be correctly forked to each diverging completion, but Prefix Grouper operates only on attention KV caches. Tree Training's SSM state routing (Section 3.2) fills this gap, making it directly applicable to SSM hybrid architectures where Prefix Grouper is not.

In summary, Tree Training strictly generalises Prefix Grouper: it handles gradient-participating shared prefixes, arbitrary tree depths (not just prompt-level sharing), and SSM hybrid architectures.

F Context Compaction and Forests

Context compaction (Claude Code's auto-compaction replaces the history with a generated summary while keeping the system prompt) does not break the tree: the post-compaction sequence [system prompt | summary | subsequent turns] branches off exactly at the end of the system prompt, so it adds a new subtree under a shared prefix node. Because the tree is built from the token sequences actually fed to the model, compaction only changes POR, never correctness; trajectories that share no tokens form a forest rooted at the empty string, and POR degrades gracefully to 0. Our SWE-smith trajectories include compaction events, which are among the branch sources of the low/medium-overlap trees in Figure 6.

G Sensitivity of Speedup beyond POR

To help practitioners estimate expected gains, we ran a controlled sweep on Qwen3-4B (bf16, single GPU, plain PyTorch reference implementation without Megatron) along the four axes below.

1. **Tree shape / POR.** Across response lengths $R \in \{64, \ldots, 512\}$ and branch counts $K \in \{2, \ldots, 16\}$, realized speedup tracks $1/(1 - \text{POR})$ monotonically:

POR	58%	70%	78%	82%
Realized speedup	1.50×	2.35×	3.35×	4.20×

2. **Batch size.** The K axis doubles as the batch-size axis (the baseline processes the K branches as one $[K, P+R]$ batch); batch size has no effect beyond its contribution to POR.
3. **Node-length distribution.** Splitting the same 1024-token prefix into nodes of 64 to 1024 tokens changes speedup by less than 1% (2.35×–2.36×): the tree mask depends on ancestor sets, not on node granularity.
4. **Partition capacity C.** Forcing a 3K-token tree into 1, 3, 5, 7 and 8 partitions reduces speedup smoothly from 2.35× to 0.73×, with no cliff. The per-partition overhead (gateway construction and a separate forward launch) is amortized by partition size, which is why the production runs of Section 4.3, with partitions of tens of thousands of tokens, stay above 95% of the bound.

Rule of thumb. Estimate POR from the trajectory batch, take $1/(1 - \text{POR})$ as the ceiling, and discount by partition count only when trees must be split into partitions that are small relative to C.

H Extending Tree Training to Other Architectures

Every adaptation in Section 3.2—mask, position IDs, SSM routing, gateways—enforces one invariant: *each token sees exactly the context it would see in a standalone per-branch forward*. This invariant is also the recipe for new architectures, and it is checkable per layer type with the gradient-verification harness of Section 4.4.

MoE and parallelism. Expert routing is per-token and order-independent, so the MoE FFN path (including large expert counts and quantized expert weights) is unaffected by tree packing; the tree only changes which attention context each token sees. Tensor, expert, data and pipeline parallelism compose as in Section 4.2. At trillion-parameter scale the engineering items are carrying varlen/-mask metadata along pipeline microbatches and setting the partition capacity C per rank from the activation budget (deeper pipelines simply lower C).

Multi-head latent attention (MLA). MLA compresses KV into a per-token latent, but attention scores are still computed per query–key pair, so the tree mask applies unchanged and the decoupled RoPE component consumes depth-based position IDs like any RoPE. The partition gateway becomes cheaper, since the cut-node state is the compressed latent rather than full per-head K/V. Multi-token-prediction heads need the same branch-point label handling as the primary head.

Sparse and compressed attention. Attention that *selects* its attended set dynamically (e.g., a lightweight indexer scoring candidate tokens) or *compresses* KV along the sequence dimension needs two additional enforcement points of the same invariant: (i) the selector's candidate pool must be restricted to the query's ancestor set, i.e., the tree mask must be applied inside the selector and not only at score time, otherwise top-k selection in the packed tree can differ from the per-branch run even when final scores are masked; (ii) sequence-dimension KV compression must not mix tokens across sibling branches, so compression states must be routed by tree parent—the same "sequential adjacency $\neq$ tree adjacency" problem solved by SSM state routing in Section 3.2. Neither changes the loss algebra (Eq. (4)) or the partitioning scheme.

Sliding-window attention. Intersect the tree mask with the window over ancestor positions; Flash-Mask's per-query start/end bounds already express this.

Other recurrent layers. Any layer that carries sequential state needs that state routed by tree parent rather than by sequence predecessor. GDN, with two coupled states (recurrent state and causal-conv context), is the harder instance of this pattern, and its adaptation (Appendix I) serves as a template.

Looped transformers. These iterate the whole stack and are left for future work.

I DFS Serialization: Implementation Details

I.1 Tree Attention Mask via FlashMask

The tree attention mask allows token i to attend to token j iff j is in an ancestor chunk of i (full access to the entire ancestor chunk) or in the same chunk with $j \leq i$ (causal within chunk). We implement this as a block-sparse mask using FlashMask Wang et al. [2025], which extends FlashAttention to support rich per-query start/end column bounds and skips masked blocks entirely.

Algorithm 1: Tree Attention Mask Construction

Input: `chunk_boundaries`, `ancestors`$[c]$ for each chunk c, total tokens N
Output: Boolean mask $M \in \{0,1\}^{N \times N}$
$M \leftarrow \mathbf{0}_{N \times N}$;
foreach *chunk* c **do**
  foreach *ancestor* $a \in$ `ancestors`$[c]$ (including c itself) **do**
    foreach *token* i *in chunk* c **do**
      if $a = c$ **then**
        $M[i,$ `c_start` $: i{+}1] \leftarrow 1$; `// causal within chunk`
      else
        $M[i,$ `a_start` $:$ `a_end`$] \leftarrow 1$; `// full access to ancestor`
return M;

I.2 Tree-Aware SSM State Routing

Standard chunk GatedDeltaNet passes the recurrent state sequentially: chunk i reads the output state of chunk $i{-}1$. Under DFS serialization, after a leaf chunk the sequence backtracks to a sibling branch, so the sequential predecessor is a sibling, not an ancestor. We replace sequential routing with tree routing: each chunk reads the output state of its *parent* chunk. `all_states`$[0]$ holds the zero initial state; `all_states`$[i{+}1]$ holds the state produced after chunk i. The mapping `parent_idx` $= \pi(c) + 1$ routes root chunks ($\pi(c) = -1$) to the zero state and non-root chunks to their parent's output state. DFS pre-order guarantees the parent state is always ready before any child requests it; sibling chunks share the same parent state tensor so their gradient contributions accumulate there automatically.

Algorithm 2: Tree-Aware GDN Forward Pass

Input: q, k, v, g, β; `chunk_boundaries`; `chunk_parent_map`; s_0
Output: `core_out`
`all_states` $\leftarrow [s_0]$;
foreach *chunk i with token range* $[c_s, c_e)$ **do**
  $q_i, k_i, v_i, g_i, \beta_i \leftarrow$ slice tensors at $[c_s, c_e)$;
  $g_{\text{cum}} \leftarrow \text{cumsum}(g_i)$;
  $D \leftarrow \exp(g_{\text{cum},j} - g_{\text{cum},k})_{\text{lower-tri}}$; `// decay mask` $[L \times L]$
  `// Within-chunk SSM correction`
  $A_{\text{raw}} \leftarrow -(k_{\beta_i} k_i^\top \odot D)$;
  $A \leftarrow \text{RowCorrect}(A_{\text{raw}}) + I$; `// Woodbury-style fix`
  $v_{\text{corr}} \leftarrow A\, v_{\beta_i}$;
  $k_{\text{cd}} \leftarrow A\,(k_{\beta_i} \odot \exp(g_{\text{cum}}))$;
  `// Cross-chunk: read from PARENT state`
  $S_{\text{cur}} \leftarrow$ `all_states`$[\pi(i)+1]$;
  $v_{\text{new}} \leftarrow v_{\text{corr}} - k_{\text{cd}}\, S_{\text{cur}}$;
  $\text{out}_i \leftarrow (q_i \odot \exp(g_{\text{cum}}))S_{\text{cur}} + (q_i k_i^\top \odot D)\, v_{\text{new}}$;
  `// Update state for children`
  $S_{\text{new}} \leftarrow S_{\text{cur}} \cdot \exp(g_{\text{cum},-1}) + k_i^\top v_{\text{new}}$;
  `all_states`.append(S_{new});
return $\text{concat}(\text{out}_0, \text{out}_1, \ldots)$;

I.3 Tree-Correct Causal Convolution

GDN layers include a causal conv1d with kernel size K_{conv}. At a DFS backtrack position, the $K_{\text{conv}}-1$ tokens immediately before a child chunk belong to a sibling branch, so using them as context is incorrect. Each chunk instead prepends its *parent*'s saved last $K_{\text{conv}}-1$ effective tokens as left context and discards them from the output. The saved context is the tail of the concatenated [`parent_ctx`; `chunk`] tensor, which propagates the full ancestor history to grandchildren.

Algorithm 3: Tree-Correct Causal Convolution

Input: X; `chunk_boundaries`; `chunk_parent_map`; conv kernel size K
Output: Y (tree-correct conv output)
`conv_states` $\leftarrow \{\}$;
foreach *chunk c with range* $[c_s, c_e)$ **do**
  $x_c \leftarrow X[:, c_s : c_e]$;
  $s_p \leftarrow$ `conv_states`.get$(\pi(c),\ \mathbf{0})$; `// parent's saved` $K-1$ `tokens`
  ext $\leftarrow [s_p\ ;\ x_c]$; `// prepend ancestor context`
  $y_{\text{ext}} \leftarrow \text{CausalConv}(\text{ext})$;
  $y_c \leftarrow y_{\text{ext}}[\,K-1 :]$; `// drop prepended tokens`
  append y_c to outputs;
  `conv_states`$[c] \leftarrow \text{ext}[\,-(K-1) :]$; `// save for children (includes ancestor history)`
return concat(outputs);

J Zero-Overhead Partition Boundaries: Implementation Details

This appendix provides the complete implementation details for the KV-cache gradient checkpointing at partition boundaries described in Section 3.3.

J.1 Data Structures

SubtreeGateway. Each cut node n_c in partition P produces one gateway for the child partition:

- $\{(\tilde{\mathbf{G}}_{n_c}^{(l)}, \tilde{\mathbf{V}}_{n_c}^{(l)})\}_l$: detached leaf KV tensors with `requires_grad=True`, shape $[1, H, e_{n_c}, d]$ per layer.

- $\{(\mathbf{G}_{n_c}^{(l)}, \mathbf{V}_{n_c}^{(l)})\}_l$: original KV slices retaining `grad_fn`.
- Float32 accumulator hooks: registered on each leaf tensor to capture gradient contributions in float32 before bfloat16 rounding; see below.
- `attn_bias` $\in \mathbb{R}^{e_{n_c}}$: additive attention bias with 0 at ancestor positions and $-\infty$ at non-ancestor positions (Section J.3).
- `pos_offset`: depth-based position of the child's first token (Section J.4).
- For SSM hybrid: $\{(\tilde{\mathbf{S}}_{n_c}^{(l)}, \tilde{\mathbf{C}}_{n_c}^{(l)})\}_l$ and their raw counterparts (Section J.7).

J.2 Forward Pass

Parent partition forward. Run DFS forward on P's token sequence with the standard tree attention mask and SSM patch (Section 3.2). The model is called with `past_key_values` $=$ GatewayCache($\tilde{\mathbf{G}}_{\text{parent}}$) if P itself is a child partition, otherwise an empty cache.

After the forward, for each cut node $n_c \in P$:

1. Slice the accumulated KV: $\mathbf{G}_{n_c}^{(l)} \leftarrow \mathbf{K}_{[:\texttt{past_len}+e_{n_c}]}^{(l)}$, which retains `grad_fn` via the concatenation inside GatewayCache.
2. Detach to create gateway leaf tensors and register float32 accumulator hooks:

$$\tilde{\mathbf{G}}_{n_c}^{(l)} = \text{detach}\Big(\mathbf{G}_{n_c}^{(l)}\Big).\,\text{requires_grad}(\texttt{True}). \tag{13}$$

Child partition forward. Run DFS forward on C's own token sequence with the *combined* attention mask:

$$M_{ij} = \begin{cases} \texttt{attn_bias}_j & j < \texttt{past_len} \\ m^{\text{tree}}_{i,\, j-\texttt{past_len}} & j \geq \texttt{past_len} \end{cases} \tag{14}$$

where m^{tree} is the standard within-subtree tree attention mask. The model receives `past_key_values` $=$ GatewayCache($\tilde{\mathbf{G}}_{n_c}$).

J.3 Ancestor-Aware Attention Bias

In a DFS-ordered partition, the gateway slice $[:\, e_{n_c}]$ may include tokens from sibling subtrees that are *not* ancestors of n_c. For example, if P's DFS sequence is $[a, b, c_1, c_2]$ where c_1, c_2 are siblings and $n_c = c_2$, the gateway includes c_1's tokens, which n_c's children must not attend to. We compute ancestor positions by traversing n_c's parent chain within P:

$$\texttt{anc}(n_c, P) = \bigcup_{n' \in \text{path}(P.\text{root}, n_c)} [s_{n'}, e_{n'}] \tag{15}$$

where $[s_{n'}, e_{n'}]$ is the token range of n' in P's DFS sequence. The attention bias is:

$$\texttt{attn_bias}_j = \begin{cases} 0 & j \in \texttt{anc}(n_c, P) \cup [0, \texttt{past_len}_P) \\ -\infty & \text{otherwise} \end{cases} \tag{16}$$

(The term $[0, \texttt{past_len}_P)$ covers P's own gateway past, which is always fully visible since it was already ancestor-filtered when P's gateway was created.)

J.4 Depth-Based Position Offset

The child partition's position IDs must match those that n_c's children would receive in a standalone per-path forward (to ensure RoPE produces identical results). Let $\text{pos}_{n_c}^{(P)}$ be the depth-based position of n_c's *first* token within P (obtained from P's `dfs_pos_ids` array). Then:

$$\texttt{pos_offset}(C) = \texttt{pos_offset}(P) + \text{pos}_{n_c}^{(P)} + |T_{n_c}| \tag{17}$$

Critically, `pos_offset`$(C) \neq$ `past_len` when P's DFS contains sibling tokens between P's first token and n_c: `past_len` counts all gateway tokens (including non-ancestor siblings), whereas (17) counts only ancestor tokens up to and including n_c.

J.5 Float32 Gradient Accumulation

When multiple child partitions share the same cut node (i.e., n_c has $k > 1$ children going to distinct child partitions), all children's backward passes accumulate gradients into the same leaf gateway tensors. Direct accumulation into the bfloat16 `.grad` attribute introduces rounding errors from repeated bfloat16 additions. We register a float32 accumulator hook on each leaf tensor:

$$\tilde{\mathbf{G}}^{(l)}\texttt{.register_hook}\Big(\lambda g : \mathrm{acc}^{(l)} \mathrel{+}= g.\mathrm{float}(),\ \text{return } \mathbf{0}\Big) \tag{18}$$

Returning $\mathbf{0}$ suppresses writes to the bfloat16 `.grad`, so the float32 accumulator captures the exact sum. The chain backward (19) then uses the float32 accumulated values. Note: even though we pass float32 gradients to `autograd.backward`, PyTorch's type-promotion rules cast them back to bfloat16 when entering a bfloat16 matmul; the hook therefore primarily eliminates *inter-child accumulation* rounding, not the in-kernel computation.

J.6 Backward Pass and retain_graph Scheduling

Processing partition P in reverse topological order:

1. For each actual leaf $\ell \in P$: compute $\mathcal{L}_\ell$ and call $\mathcal{L}_\ell$`.backward(retain_graph =` `True`$)$.
2. For each unique cut node n_c of P (deduplicated to avoid double-counting when n_c has multiple children): chain gradients back via:

$$\texttt{autograd.backward}\Big(\Big\{\mathbf{G}_{n_c}^{(l)}\Big\}_l,\ \Big\{\nabla_{\tilde{\mathbf{G}}_{n_c}^{(l)}}\Big\}_l\Big) \tag{19}$$

   with `retain_graph` $=$ `True` for all but the last operation, `False` for the last.

Total backward traversals of P's computation graph $= |\{\ell \in P : \ell \text{ is a leaf}\}| + |\text{unique cut nodes of } P|$. The graph is freed after the last traversal.

J.7 SSM State Gateway

State extraction. The existing DFS SSM forward (`torch_chunk_gated_delta_rule_tree_varlen`) maintains an `all_states` list where `all_states[i+1]` is the recurrent state after chunk i. We extend it with a `capture_states_at` argument: after computing `new_state` for chunk c, if $c \in$ `capture_states_at`, store `all_states[c+1]` (which has `grad_fn`). Similarly, `_tree_correct_conv_varlen` is extended to return `conv_states[c]` (the $K-1$ conv context after chunk c) at requested positions.

State injection. For the child partition's SSM forward, we inject gateway states via two mechanisms:

- *Recurrent state*: pass $\tilde{\mathbf{S}}_{n_c}^{(l)}$ as `initial_state` to `torch_chunk_gated_delta_rule_tree_varlen`. Root chunks (those with `parent_map` $= -1$) read from `all_states[0]` $=$ `initial_state`, matching the gateway.
- *Conv context*: pre-populate `conv_states[-1]` $= \tilde{\mathbf{C}}_{n_c}^{(l)}$ before calling `_tree_correct_conv_varlen`. Root chunks look up `conv_states.get(-1, None)` as their parent context, finding the gateway value.

This design requires no structural changes to the SSM forward; the injection is entirely through the existing lookup mechanism.

Gradient chaining. The raw tensors $\mathbf{S}_{n_c}^{(l)}$ and $\mathbf{C}_{n_c}^{(l)}$ (with `grad_fn`) are included in the `autograd.backward` call alongside the KV tensors. All KV and SSM gradients are chained in a single call per cut node, with a shared `retain_graph` flag.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state the three contributions: the mathematical equivalence derivation, DFS serialization with model-layer adaptations, and Redundancy-Free Tree Partitioning. All claims are supported by the theoretical derivations and experiments in the paper.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses scope limitations in Section 3.1: objectives that model joint distributions across multiple paths simultaneously (rather than evaluating each path independently) are outside the current framework. The conclusion notes that extension to non-standard architectures such as looped transformers is left for future work. The experiments are conducted on the Qwen model family; broader architectural coverage is a direction for future work.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: The paper provides a complete algebraic derivation showing that the tree loss equals the sep-avg loss (Section 3.1). Assumptions are explicitly stated: per-token objectives are additive across tokens and paths.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper specifies the models (Qwen3-30B, Qwen3-32B), hardware (64 NVIDIA Hopper GPUs), training framework (Megatron-Core), baseline (Sequence Packing), and key implementation details in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We publicly release the reference implementation (DFS serialization and tree attention mask, tree-aware GDN state and conv-context routing, redundancy-free partitioning with differentiable gateways) and the gradient-verification harness with reproduction commands at `https://github.com/Jinghui-Wang/tree-training` (Section 4.2). Code for gradient correctness verification across multiple architectures (Qwen3, Qwen3.5, Llama-3.2) is provided in the supplemental material (Appendix A). The end-to-end training speedup experiments rely on an internal Megatron-Core integration that is not publicly released; however, the algorithm is fully described in the paper and appendices with sufficient detail for reimplementation.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The experimental setup section specifies hardware (64 NVIDIA Hopper GPUs), training framework (Megatron-Core), and baseline (Sequence Packing). Model architectures are cited.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.

- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [N/A]

   Justification: The speedup results are deterministic ratios of wall-clock time between methods on the same hardware; error bars are not applicable. The loss alignment experiments report per-step deviations which are shown in figures.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper specifies that all experiments were conducted on clusters of 64 NVIDIA Hopper GPUs using Megatron-Core.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This paper presents a training efficiency method with no ethical concerns. No human subjects, sensitive data, or dual-use risks are involved.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This paper is a training efficiency method. It reduces computational cost of LLM training but introduces no direct negative societal impact beyond what applies to LLM training in general.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: No new models or datasets are released in this paper.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All models (Qwen3, Qwen3.5), datasets (SWE-smith), and frameworks (Megatron-Core, FlashAttention, FlashMask, OR-Tools) used are properly cited.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not introduce new datasets or model checkpoints.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: No human subjects are involved.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are the subject of study (the models being trained), not a component of the research methodology itself.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.