LOOP-PP: PIPELINE PARALLELISM FOR LOOPED TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped Transformers increase effective model depth and capacity by recursively applying shared blocks, but their recurrent execution is poorly matched to conventional pipeline parallelism (PP), which assumes a one-pass feed-forward execution order. When recurrent blocks span multiple pipeline stages, outputs from later stages feed back to earlier ones, disrupting conventional PP's fixed compute and communication orders. To this end, we present *Loop-PP*, the first pipeline-parallel training system designed for Looped Transformers. Its core, the *Loop-Aware Pipeline Compiler*, first makes repeated logical visits and recurrent dependencies explicit in a scheduling-independent *Semantic Graph*, then synthesizes a *Global Compute Order* under these semantic constraints, and finally compiles cross-stage dependencies into *Per-Rank Action Streams* that overlap communication with computation. Across 8B–60B models, Loop-PP maintains broadly comparable throughput to compute-matched baseline with interleaved 1F1B while reducing memory by up to 32.8%. Under an equal resource budget, it improves the best measured throughput by up to 12.7%.

1 INTRODUCTION

Recent advances in large language models (LLMs) have been driven primarily by scaling model parameters (Kaplan et al., 2020; Brown et al., 2020), data (Hoffmann et al., 2022), and test-time computation (Snell et al., 2024). Looped Transformers or depth-recurrent Transformers offer a different path by recursively applying the same blocks, increasing effective model depth and capacity without adding parameters (Dehghani et al., 2019; Lan et al., 2020; Geiping et al., 2025; Zhu et al., 2025). Recent studies have shown that such models hold promise for reasoning, in-context learning, and test-time scaling (Saunshi et al., 2025; Yang et al., 2024; Alabdulmohsin & Zhai, 2025).

Existing work, however, has focused primarily on model architectures, leaving the systems support for large-scale depth-recurrent training underexplored. As models outgrow single-device memory, their parameters must be distributed across devices. Pipeline parallelism (PP) partitions model layers into stages placed across devices and pipelines micro-batches through them, offering favorable communication volume and arithmetic intensity for large-scale training (Huang et al., 2019; Narayanan et al., 2021a; Shoeybi et al., 2019; Rajbhandari et al., 2020).

However, conventional PP schedules, *e.g.*, Megatron's interleaved 1F1B (Narayanan et al., 2021b), are designed for linear feed-forward execution, where each layer is used once in a forward pass. Looped Transformers instead revisit shared blocks spanning multiple stages, introducing loop-carried feedback from later to earlier stages that breaks the compute-readiness and communication-order assumptions of native interleaved 1F1B (Figure 1). Without global rescheduling, the direct adaptation of conventional PP is limited to independent stage-local repetition. Consider a recurrent region repeated for $\mathcal{T}$ steps and partitioned across P stages, where $\mathcal{F}_p$ denotes the recurrent computation assigned to stage p. Such stage-local repetition yields

$$\underbrace{\mathcal{F}_{P-1}^{\mathcal{T}} \circ \cdots \circ \mathcal{F}_0^{\mathcal{T}}}_{\text{independent stage-local repetition}}, \tag{1}$$

whereas the looped model requires

$$\underbrace{(\mathcal{F}_{P-1} \circ \cdots \circ \mathcal{F}_0)^{\mathcal{T}}}_{\mathcal{T} \text{ end-to-end recurrent steps}}. \tag{2}$$

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

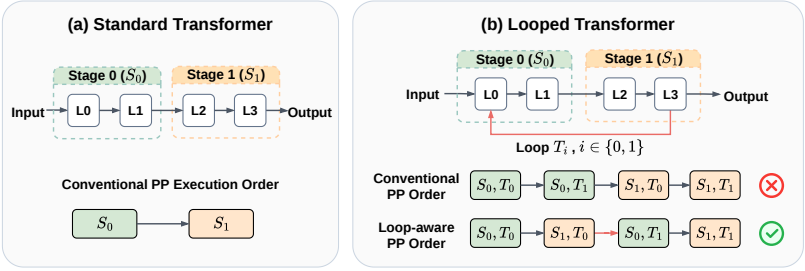

Figure 1: Pipeline execution of (a) Standard Transformer and (b) Looped Transformer.

Here, the superscript $\mathcal{T}$ denotes repeated function composition. Thus, simply inserting an independent local loop into each pipeline stage is insufficient.

A straightforward alternative is to physically unroll the $\mathcal{T}$ recurrent visits into separate pipeline chunks and apply conventional PP scheduling. However, using independent parameters changes the model semantics, while tying the expanded copies incurs additional parameter storage and synchronization overhead. The challenge is therefore to preserve exact recurrence semantics and a single physical copy of shared parameters while retaining the concurrency of native pipeline execution.

To this end, we present *Loop-PP*, the first pipeline-parallel training system for looped Transformers. Its core is a *Loop-Aware Pipeline Compiler* which translates recurrent model semantics into static pipeline execution plans while preserving a single physical copy of shared parameters. First, *Semantic Graph Construction* logically unrolls the recurrent computation into distinct visits to shared physical model regions and captures their cross-step dependencies, yielding a scheduling-independent *Semantic Graph*. Second, *Native-Guided Schedule Synthesis* uses native interleaved 1F1B as a compute template and constrains it with these semantic dependencies, producing a *Global Compute Order*. Third, *Communication-Aware Stream Compilation* turns cross-stage dependencies into ordered communication actions and schedules their initiation and completion around dependency-ready compute to expose communication overlap, producing a static *Per-Rank Action Stream* for each rank. We further introduce system optimizations to reduce communication and runtime overhead. In summary, our main contributions are:

- We identify the mismatch between conventional pipeline parallelism and depth-recurrent execution, and, to our knowledge, present the first pipeline-parallel training system designed for looped Transformers.
- We develop Loop-PP, a loop-aware pipeline compiler that preserves recurrent semantics and parameter sharing while adapting native interleaved 1F1B scheduling and communication to repeated cross-stage execution.
- Across 8B–60B models, Loop-PP maintains broadly comparable throughput to compute-matched baseline with interleaved 1F1B while reducing peak memory by up to 32.8%. Under an equal resource budget, it improves the best measured throughput by up to 12.7%.

2 Background and Motivation

Looped Transformers Looped models increase effective depth by repeatedly applying shared blocks (Dehghani et al., 2019; Geiping et al., 2025; Zhu et al., 2025; Saunshi et al., 2025). Existing models vary in recurrence scope, from intermediate regions (Geiping et al., 2025) to nearly the entire Transformer backbone (Zhu et al., 2025). To capture both cases, we decompose the model into a non-recurrent prelude $\mathcal{P}$, a recurrent region $\mathcal{R}$, and a non-recurrent coda $\mathcal{C}$. For full-backbone recurrence, $\mathcal{P}$ and $\mathcal{C}$ contain only boundary computation such as the embedding and output head.

Let $a = \mathcal{P}(x; \theta_{\mathcal{P}})$ denote the prelude output with input x and parameters $\theta_{\mathcal{P}}$. Starting from $h_0 = a$, the recurrent region is applied for $\mathcal{T}$ steps:

$$h_{t+1} = \mathcal{R}(h_t, a; \theta_{\mathcal{R}}), \qquad t = 0, \ldots, \mathcal{T} - 1, \tag{3}$$

followed by $y = \mathcal{C}(h_{\mathcal{T}}; \theta_{\mathcal{C}})$. The parameters $\theta_{\mathcal{R}}$ are shared across recurrent steps, and the dependence of h_{t+1} on h_t captures the recurrent state carried across steps. The second argument a is optional: some models omit it and update from h_t alone, while others (Geiping et al., 2025) inject

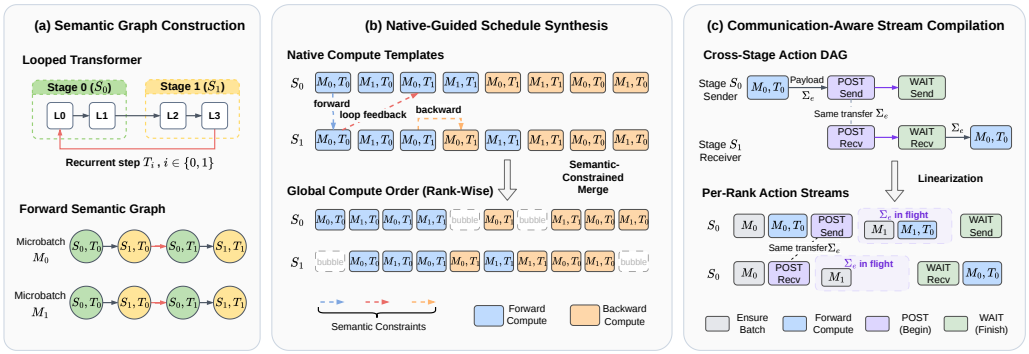

Figure 2: Overview of the Loop-Aware Pipeline Compiler. S_p denotes pipeline stage p, T_t denotes recurrent step t, and M_m denotes micro-batch m.

a at every step, making the unrolled computation a DAG rather than a linear chain. When these dependencies span pipeline stages, PP must communicate recurrent states and, when present, injected inputs in addition to the hidden activations transferred in conventional feed-forward execution. Conventional PP assumes a single feed-forward visit to each model chunk, whereas a recurrent region spanning stages requires cross-stage feedback and side dependencies. Therefore, we need to design a new pipeline parallel scheduling strategy for looped models.

3 Loop-Aware Pipeline Compiler

The *Loop-Aware Pipeline Compiler* translates a depth-recurrent model into deterministic computation and communication sequences for pipeline-parallel training. It takes as input the recurrent model structure, the number of micro-batches M, and PP/VPP placement, and produces static Per-Rank Action Streams for each pipeline rank through the following three stages, as shown in Figure 2.

3.1 Semantic Graph Construction

A looped model repeatedly invokes the same physical blocks, but each invocation consumes a different intermediate state and therefore has a distinct logical identity. Figure 2 (a) makes this unrolling explicit: T_0 and T_1 on the same pipeline stage S_p are two logical visits to the same physical module, while different micro-batches M_m form disjoint graph copies. The compiler unrolls the fixed $\mathcal{T}$ recurrent steps into a forward *Semantic Graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. For each micro-batch, the dataflow has the form

$$\mathcal{P} \to \mathcal{R}^{(0)} \to \cdots \to \mathcal{R}^{(\mathcal{T}-1)} \to \mathcal{C}, \tag{4}$$

where each $\mathcal{R}^{(t)}$ is the within-step DAG of the recurrent region.

Concretely, the compiler first partitions the prelude, recurrent region, and coda into segments by PP/VPP placement and creates one semantic node v for each visit. Each node $v \in \mathcal{V}$ records its micro-batch M_m, recurrent step T_t, computation role, PP stage S_p, and VPP placement. Prelude and coda are visited once per micro-batch, whereas the recurrent region contributes a new set of logical nodes at every t. The compiler then connects computations within a step by main edges, connects the exit of step t to the entry of step $t+1$ by feedback edges (the cross-stage recurrence in Figure 2), and routes loop-invariant or auxiliary state a to its consumers by injection edges. An edge $e = (u, v, \Sigma_e)$ carries an ordered tensor bundle Σ_e from producer u to consumer v that may contain multiple tensors, *e.g.*, auxiliary state a and main hidden activation.

Unrolling creates a distinct logical visit for each recurrent step, so a feedback edge connects step t only to step $t+1$ and the graph is acyclic for finite $\mathcal{T}$. Injection edges may additionally fan out from the prelude to several recurrent nodes, so $\mathcal{G}$ need not be a single chain. The graph specifies what to compute, which dependencies to respect, and on which S_p each node runs, but it does not yet choose an execution order. Backward compute is derived later from the reversed dependencies. Appendix A gives the complete node and edge definitions, the construction algorithm, and the semantic-equivalence argument.

3.2 NATIVE-GUIDED SCHEDULE SYNTHESIS

The Semantic Graph defines a partial order over model computations, but it does not specify which operation to choose when several are simultaneously ready. Any topological execution preserves model semantics, yet an arbitrary choice can leave pipeline stages idle. Efficient scheduling must satisfy the recurrent dependencies while retaining the compute cadence of established pipeline schedules (Huang et al., 2019; Narayanan et al., 2021a; Qi et al., 2024b). We therefore use native interleaved 1F1B (Narayanan et al., 2021b) as a compute template and constrain it with the Semantic Graph. This section orders forward and backward computation without introducing communication.

Native Compute Template. For each PP stage, the compiler treats every visit to a layer-bearing segment on S_p as a *logical compute chunk*: recurrent visits are distinct chunks but still invoke the same physical parameters. It then applies Megatron's interleaved 1F1B rule to build a communication-free reference sequence for each stage,

$$\mathcal{N}_p^{\text{nat}} = [(\phi_i, m_i, c_i)]_{i=0}^{J_p-1}, \qquad \phi_i \in \{F, B\}, \tag{5}$$

where J_p is the number of native compute slots on stage p and Appendix B gives the complete slot construction. Slot i records the phase ϕ_i (forward or backward), the micro-batch m_i assigned to that slot, and the logical compute chunk c_i. Figure 2 (b) illustrates that T_0 and T_1 on the same S_p are two logical visits, while M_m indexes the forward/backward slots of different micro-batches. Recurrence therefore expands the number of logical slots, not the number of physical layers.

Semantic-constrained Merge. The native templates specify the desired interleaved 1F1B cadence, but do not encode recurrent dependencies in the Semantic Graph $\mathcal{G}$. Executing them independently may therefore start a semantic consumer before its input has been produced. To make these dependencies explicit, the compiler expands each native slot into the semantic nodes attached to its logical compute chunk and represents every node v by two phased semantic operations, forward operation (F, v) and backward operation (B, v). These semantic operations, rather than the abstract native slots, are the objects ultimately ordered by the scheduler.

The compiler then combines the semantic dependencies in $\mathcal{G}$ with the native order $\mathcal{N}_p^{\text{nat}}$ into a phased constraint graph,

$$\mathcal{H} = \left(\{(\phi, v) \mid \phi \in \{F, B\},\ v \in \mathcal{V}\}, \mathcal{E}_{\text{nat}} \cup \mathcal{E}_F \cup \mathcal{E}_B \cup \mathcal{E}_{FB}\right), \tag{6}$$

For every semantic edge $(u, v) \in \mathcal{E}$, $\mathcal{E}_F$ contains $(F, u) \prec (F, v)$ and $\mathcal{E}_B$ contains the reverse-mode constraint $(B, v) \prec (B, u)$. For every node $v \in \mathcal{V}$, $\mathcal{E}_{FB}$ contains $(F, v) \prec (B, v)$. The Semantic Constraints highlighted in Figure 2 illustrate these model-derived dependencies. In particular, the illustrated loop feedback is an edge in $\mathcal{E}_F$ requiring the exit forward of step t to precede the entry forward of step $t + 1$. On each stage, native 1F1B requires operations to follow their order in $\mathcal{N}_p^{\text{nat}}$. After mapping its slots to phased semantic operations, the compiler encodes this order by adding an edge between each consecutive pair, and their union across all stages forms $\mathcal{E}_{\text{nat}}$.

Next, the compiler computes a deterministic linear extension of $\mathcal{H}$ with a priority-guided Kahn algorithm (Kahn, 1962). All zero-indegree operations form a global ready heap, so an operation becomes eligible only after all of its semantic and stage-local native predecessors have been emitted. Among them, the compiler selects the lexicographically smallest $(r_{\text{nat}}, r_{\text{phase}}, \text{id})$, where r_{nat} is its position in the native order of its stage, $r_{\text{phase}} = 0$ for forward and 1 for backward, and the semantic-node identity provides deterministic tie breaking. Because $\mathcal{E}_{\text{nat}}$ chains the operations on each stage, the algorithm chooses only among the ready frontiers of different stages without changing any stage-local native order. Emitting all operations produces the *Global Compute Order*

$$\mathcal{O} = [(\phi_0, v_0), \ldots, (\phi_{N_{\text{op}}-1}, v_{N_{\text{op}}-1})], \qquad \phi_i \in \{F, B\}, \quad v_i \in \mathcal{V}, \quad N_{\text{op}} = 2|\mathcal{V}|, \tag{7}$$

which contains every semantic forward and backward operation exactly once. As illustrated by the rank-wise view in Figure 2(b), the Global Compute Order preserves each rank's native compute order without serializing independent operations across ranks at runtime. Native-rank priority is applied only among zero-indegree operations in $\mathcal{H}$, so it never bypasses a semantic dependency. Appendix B gives the detailed construction.

Under balanced chunk costs, the structural bubble matches that of compute-matched interleaved 1F1B (Appendix E.1). Note that structural bubbles characterize only the compute schedule. The actual execution time also depends on computation-communication overlap, which we address next.

3.3 COMMUNICATION-AWARE STREAM COMPILATION

The Global Compute Order specifies the computation order, but it does not determine communication actions. This section introduces communication dependencies and compiles the compute and communication actions into a deterministic executable stream for every pipeline rank.

Communication Planning. The compiler classifies each semantic edge by the PP placements of its endpoints: a local edge whose endpoints share a stage remains an in-memory data dependency, whereas a remote edge spanning stages induces a planned transfer. For a remote edge $e = (u, v, \Sigma_e)$, a forward transfer moves Σ_e from the stage owning u to the stage owning v, whereas a backward transfer moves $\nabla\Sigma_e$ from the stage owning v to the stage owning u. A transfer is represented as

$$\gamma = (e, \phi, s_{\mathrm{src}}, s_{\mathrm{dst}}, \Sigma_e^{\phi}), \qquad \phi \in \{F, B\}, \tag{8}$$

where ϕ denotes the forward or backward direction, s_{src} and s_{dst} identify the communicating stages, and $\Sigma_e^F = \Sigma_e$ and $\Sigma_e^B = \nabla\Sigma_e$ denote the corresponding payload schemas. Remote transfers are ordered according to the forward and backward subsequences of $\mathcal{O}$. The resulting *Communication Plan* specifies what data moves between which stages and in what order, as detailed in Appendix C.

Cross-Stage Action DAG. To represent communication–computation overlap, the compiler separates the initiation and completion of every transfer. A POST submits the nonblocking send or receive bundle and returns before the transfer necessarily completes, whereas the corresponding WAIT confirms the transfer's completion. Thus, it expands $\mathcal{O}$ and the Communication Plan into a *Cross-Stage Action DAG* containing forward/backward COMPUTE, nonblocking POST, WAIT, and ENSUREBATCH for micro-batch preparation.

For a transfer γ associated with $e = (u, v, \Sigma_e)$, let $(x, y) = (u, v)$ when $\phi = F$ and $(x, y) = (v, u)$ when $\phi = B$. The Action DAG enforces

$$\begin{aligned} \mathrm{Compute}^{\phi}(x) &\prec \mathrm{POST}_{\mathrm{src}}(\gamma), \\ \{\mathrm{POST}_{\mathrm{src}}(\gamma), \mathrm{POST}_{\mathrm{dst}}(\gamma)\} &\prec \{\mathrm{WAIT}_{\mathrm{src}}(\gamma), \mathrm{WAIT}_{\mathrm{dst}}(\gamma)\}, \\ \mathrm{WAIT}_{\mathrm{dst}}(\gamma) &\prec \mathrm{Compute}^{\phi}(y). \end{aligned} \tag{9}$$

These constraints ensure that a payload is produced before transfer initiation, both endpoints initiate matching communication before completion, and the receiver completes the transfer before consuming the payload. The graph also retains semantic compute dependencies and communication-order and resource-safety constraints required for execution. Crucially, the Cross-Stage Action DAG specifies which action reorderings and overlaps are legal. Actions with no dependency path between them may change relative order during linearization. In particular, independent computation may be placed between POST and WAIT to overlap with in-flight communication. Figure 2(c) illustrates how a remote dependency expands into POST and WAIT actions at its sender and receiver.

Native-Aware Communication Linearization. Because the Cross-Stage Action DAG defines only a partial order, it generally admits multiple legal executions. Linearization selects deterministic Per-Rank Action Streams from these executions while retaining the native interleaved 1F1B compute cadence and exposing communication overlap.

Ranks cannot make this choice independently: selecting sends and receives in different orders may produce inconsistent communication matching. The compiler therefore first fixes a global matching order for remote transfers. Among ready transfers, it preferentially groups opposite-direction transfers between the same stages into a bidirectional communication *epoch*, following the P2P structure of interleaved pipeline execution (Narayanan et al., 2021b). A transfer without a legal partner forms a singleton epoch. An epoch represents one globally agreed communication-matching decision, not a runtime barrier. Once their order is fixed, both endpoints initiate matching operations consistently without having to reach an epoch simultaneously.

The compiler then projects the fixed epoch order onto each stage and linearizes its local actions using three rules: (i) post the next dependency-ready and resource-feasible epoch as early as possible; (ii) otherwise, select the earliest ready COMPUTE in that stage's native compute order; and (iii) emit a WAIT only when a consumer requires the result or an in-flight transfer blocks further progress. Consequently, POSTs open communication windows early, WAITs close them only when

necessary, and independent native-preferred computation fills the interval between them. The Action DAG guarantees legality, the epoch order guarantees consistent cross-rank matching, and these rules determine the concrete local order. As illustrated in the lower part of Figure 2(c), the result is

$$\{\mathcal{Q}_s\}_{s=0}^{P-1}, \tag{10}$$

where $\mathcal{Q}_s$ is the *Per-Rank Action Stream* for stage s. All streams are compiled before execution and the runtime follows them directly. Appendix C gives the detailed construction and proves that the compiled streams preserve the Semantic Graph dependencies and yield PP-transport deadlock freedom under explicit backend-progress assumptions.

Implementation Optimizations The three compilation stages above produce a correct and executable plan, but steady-state training can still incur avoidable communication and compilation overheads. First, a uniform payload may carry loop auxiliary tensors even when remote consumers do not require those fields. *Payload Localization* therefore rewrites each payload schema according to its actual consumers and transmits only the fields required by remote computation. Second, the Global Compute Order, Communication Plan, and Per-Rank Action Streams can be reused across training steps. The compiler therefore uses *compilation caching* to reuse these results, keyed by the Semantic Graph and execution configuration. On a cache hit, steady-state training directly executes the validated local streams. Appendix D describes the implementation details.

4 Experiments

4.1 Experimental Setup

Models. We evaluate three full-loop MoE models based on Megatron-LM (Narayanan et al., 2021b): 8B-A1B, 30B-A3B, and 60B-A5B with 32, 48, and 48 unique layers. For looped models, the complete layer stack is recurrently executed for $\mathcal{T} = 2$ steps, giving effective depths of 64, 96, and 96 layers. The model names refer to the unique physical parameterization rather than the parameter count of an untied expansion. Unless otherwise specified, experiments use 32 GPUs, sequence length 4,096, and $\mathcal{T} = 2$. We report end-to-end training throughput in tokens per second and peak GPU memory in GiB. We place complete model and training configurations in Appendix G and validate the correctness of Loop-PP against a PP1 recurrent oracle in Appendix H.

Baselines. We compare three strategies with Loop-PP. *Unrolled* materializes the same effective depth with independent parameters and schedules it with Megatron's interleaved 1F1B. It matches the computation of the looped models but not their parameter-sharing semantics, serving as an optimistic native-schedule reference. *Tied* uses the same physical expansion but ties the weights across recurrent visits, preserving the looped model's semantics. After backward, gradients are reduced, only one replica is optimized, and the updated weights are copied to all replicas. This preserves interleaved 1F1B scheduling at the cost of replicated parameters and additional synchronization. *Lockstep* executes the same Semantic Graph as Loop-PP with a conservative, dependency-respecting 1F1B schedule that bounds the number of live micro-batches. It serves as a correctness-oriented reference without Loop-PP's optimized scheduling and communication. See Appendix G.2 for details.

4.2 Overall Performance

Comparison under matched configuration. We evaluate Loop-PP and the baselines on different model sizes and PP degrees, and report throughput and peak memory in Figure 3. For a given model and PP degree, all methods use the same training configuration. The settings may slightly differ across models and PP degrees, as detailed in Appendix G.1.

Loop-PP uses less peak memory than Unrolled and Tied at every plotted point, saving up to 32.8% memory. At 60B PP2, Unrolled, Tied, and Lockstep run out of memory, while Loop-PP completes. Loop-PP's throughput is competitive with the expanded native schedules and depends on pipeline granularity. At PP2, Loop-PP is 12.4% faster than Unrolled on 30B, but at PP8 it is slower. We will further analyze this reversal in Section 4.3. Tied is close to Unrolled but usually slightly slower, due to the extra gradient/parameter synchronization. Relative to Lockstep, Loop-PP is $1.9\times$ faster at PP2 and more than $6\times$ faster at PP8 on 30B. Thus Loop-PP approaches Unrolled in throughput, remains substantially faster than the conservative Lockstep baseline, and retains a large memory advantage.

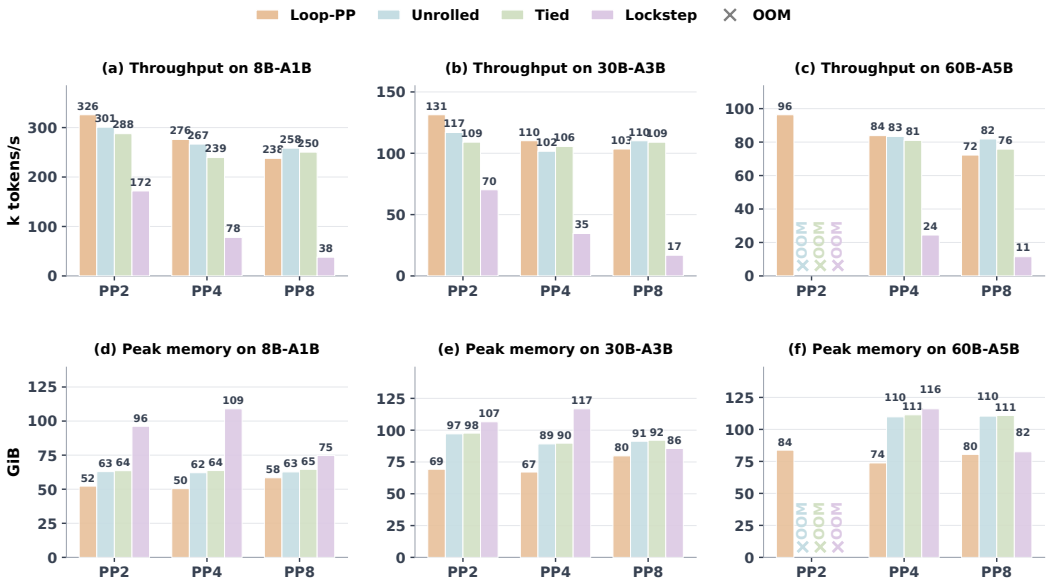

Figure 3: Training throughput and peak memory for three MoE models at PP2, PP4, and PP8.

Loop-PP achieves best performance under an equal resource budget. Unlike the previous comparison with the configuration fixed for each model and pipeline degree, we keep the model (Dense 14B or 30B MoE), the global batch size (512), and the 32-GPU budget fixed, and let Loop-PP and Unrolled choose different configurations, *e.g.*, PP, VPP, micro-batch size, to saturate the 32 GPUs. The model configurations and the per-model search traces are given in Appendix G.10. Table 1 reports the best measured throughput of each method. On 30B-A3B, Loop-PP improves the best measured throughput by 12.7%, using only slightly more peak memory at that throughput-optimal point. This shows that Loop-PP's memory saving is usable in practice: the saved memory can be converted into a more efficient operating point, so Loop-PP trains faster than the compute-matched Unrolled baseline under the same budget.

Table 1: Best throughput under an equal 32-GPU resource budget. Memory is peak allocated GiB.

Model	Loop-PP	Unrolled	Speedup	Loop mem.	Unrolled mem.
Dense 14B	96.6k	87.0k	1.110×	113.7	115.7
MoE 30B	170.7k	151.5k	1.127×	117.7	112.7

4.3 In-Depth Analysis

Understanding the PP8 Performance Cliff To understand why Loop-PP falls behind at PP8, we profile matched 30B Loop-PP and Unrolled runs on 16 GPUs. We decompose iteration time T_{step} into three components (Appendix G.4). T_{local} is the service time of local forward and backward actions, including computation and work internal to MoE actions. T_{exposed} is pipeline time not covered by local actions and can be viewed approximately as the exposed pipeline cost. T_{tail} contains gradient finalization, optimizer work, and other costs outside the pipeline schedule. Figure 4(a–b) presents this decomposition on PP2 and PP8. We additionally compare Loop-PP and Unrolled across VPP1 and VPP2 to study the effect of compute-chunk granularity in Figure 4(c).

In Figure 4(a–b), Loop-PP trades lower local and tail costs for a higher exposed pipeline cost. Its smaller T_{local} and T_{tail} are consistent with reusing shared parameters, while recurrent cross-stage dependencies increase T_{exposed}, the exposed recurrent pipeline cost. At PP2, Loop-PP has a thick compute chunk, so its shorter local computation and tail outweigh the additional exposed time. At PP8, thinner compute chunks expose recurrent dependencies more frequently, increasing communication intensity and leaving less independent computation to hide it (Appendix E.2). Consequently, the exposed-time penalty overwhelms the local and tail savings, making Loop-PP slower than Unrolled. Figure 4(c) further confirms this, where changing from VPP2 to VPP1 doubles the Loop-PP chunk from three to six unique layers and raises its throughput from 64.7k to 71.3k tokens/s, narrowing the gap to Unrolled. Together with the analysis in Appendix G.5, which attributes 99.98% of

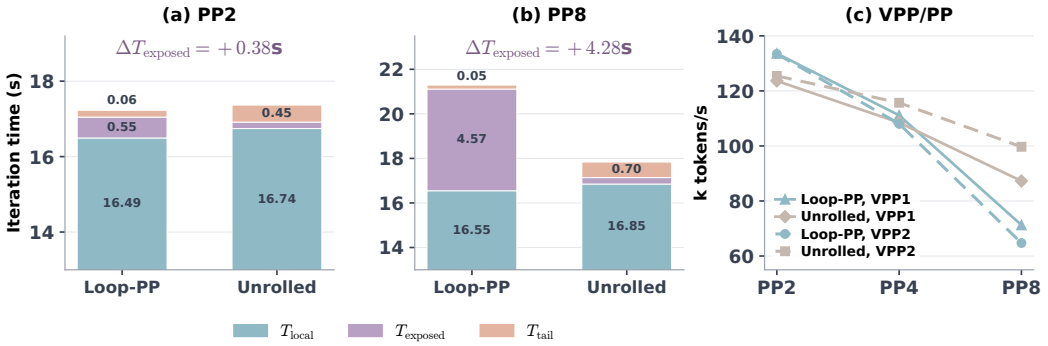

Figure 4: Diagnosing the PP8 performance cliff from 30B MoE profiling. (a–b) Iteration decomposition at PP2 and PP8. (c) Throughput under different VPP/PP.

the measured PP8 receive-wait time to dependency-limited waits, this result shows that sufficiently thick compute chunks are important for amortizing the recurrent pipeline overhead, and Loop-PP should avoid overly fine pipeline partitions.

Loop-PP across Dense and MoE Workloads. To study workload sensitivity, we replace the MoE blocks in the 30B model with dense FFNs while otherwise retaining its architecture and activation settings, and evaluate both variants at PP2 and PP4 on 32 GPUs (Appendix G.8). As shown in Figure 5, Loop-PP reduces local-action and tail times but increases exposed pipeline time across all four settings. The Dense results show that the local savings are not solely due to MoE load balancing and may arise from parameter and runtime-object reuse. The savings are larger on MoE, making Loop-PP 7.2% and 0.9% faster at PP2 and PP4. Phase profiling further shows that Loop-PP reduces all-to-all event duration from 4.195 s to 3.451 s at PP2 and from 4.498 s to 3.779 s at PP4. This association may reflect runtime effects of parameter reuse, such as lower memory pressure and more favorable communication-buffer behavior. Loop-PP therefore benefits MoE workloads more, where larger local savings better amortize the recurrent pipeline overhead. Additional workload sweeps are reported in Appendix G.9.

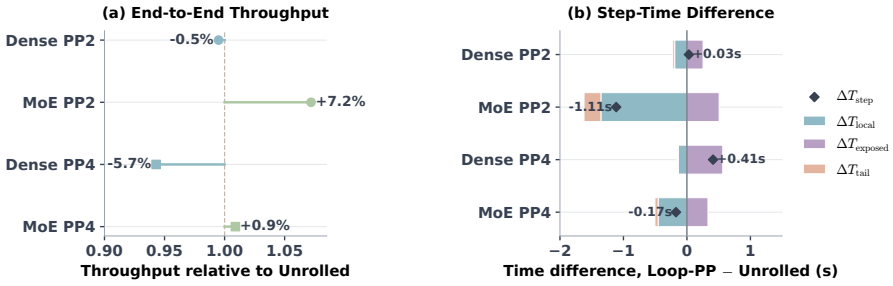

Figure 5: Comparison of Dense and MoE models. (a) Throughput relative to Unrolled. (b) Step-time difference $\Delta T_{\text{step}} = T_{\text{step}}^{\text{Loop-PP}} - T_{\text{step}}^{\text{Unrolled}}$. The diamonds denote the total difference.

Impact of Recurrent Structure We study recurrent structure on the 8B model by varying the number of recurrent steps, the recurrent fraction, and the recurrent layout. Figure 6(a) compares PP2 and PP4 as the number of full-loop steps increases from two to four. As $\mathcal{T}$ grows to four, Loop-PP's relative throughput weakens at both PP degrees and falls behind Unrolled at PP4, showing that more recurrent traversals incur a larger structural tax. Figure 6(b) fixes $\mathcal{T} = 2$ and varies the recurrent fraction from 25% to 100% at PP2 and PP4. Performance varies non-monotonically with the recurrent fraction: the 50% and 100% configurations outperform the 25% and 75% configurations. This shows that performance depends on how the recurrent region maps to pipeline stages, rather than on recurrent fraction alone. Figure 6(c) compares three layouts at the same 64-layer effective depth. The full-loop 32×2 and $8+16 \times 3+8$ layouts are close at PP4, while 32×2 leads at PP2. The $12+8 \times 5+12$ layout incurs more recurrent traversals and cross-stage feedback, reducing opportunities to hide communication and increasing dependency stalls. Overall, more recurrent traversals and placement-sensitive cross-stage dependencies amplify the structural tax, favoring pipeline layouts with sufficiently thick compute chunks. See Appendix G.6 for detailed analysis.

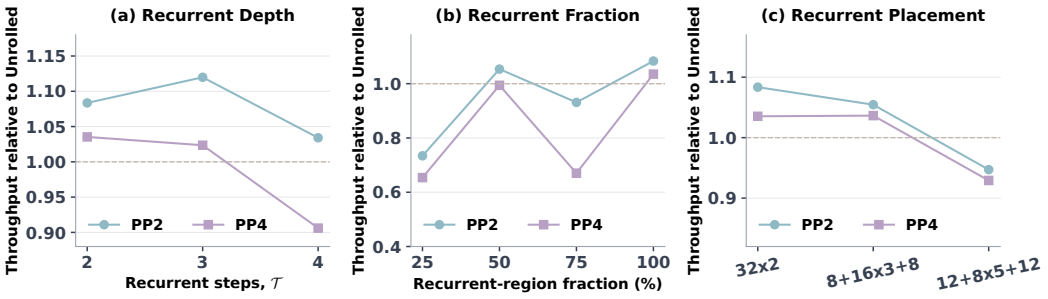

Figure 6: Sensitivity to recurrent structure on the 8B workload. (a) Throughput relative to Unrolled as full-loop recurrent depth increases. (b) Throughput relative to Unrolled across recurrent-region fractions at $\mathcal{T} = 2$. (c) Three prelude–recurrent–coda layouts with the same 64-layer effective depth.

5 Related Work

Pipeline Parallelism Conventional PP partitions layers across devices for feed-forward execution. GPipe (Huang et al., 2019) pipelines micro-batches, while PipeDream-2BW (Narayanan et al., 2021a) and Megatron-LM (Narayanan et al., 2021b) improve utilization through 1F1B scheduling, weight buffering, and interleaved virtual stages. Subsequent work reduces pipeline bubbles through finer-grained or asynchronous scheduling (Qi et al., 2024b; Ajanthan et al., 2025; Jung et al., 2026; Zmushko et al., 2026; Chen et al., 2026a), improves memory efficiency (Kim et al., 2023; Qi et al., 2024a; Wan et al., 2025), or automates partitioning and scheduling (Jhoo et al., 2025). GraphPipe (Jeon et al., 2025) and PULSE (Sun et al., 2026) extend PP to DAG-structured networks and skip-aware diffusion U-Nets, respectively, but do not directly address repeated execution of shared physical chunks with cross-stage recurrent feedback. Other work pipelines computation along token, sequence, or time dimensions (Li et al., 2021; Zhao et al., 2026; Bao et al., 2026), which is orthogonal to our depth-recurrent setting.

Looped Transformers Looped Transformers increase effective depth by repeatedly applying shared layers. Building on early weight-tied architectures (Dehghani et al., 2019; Lan et al., 2020), recent models reuse intermediate blocks or entire layer stacks (Geiping et al., 2025; Zhu et al., 2025; Saunshi et al., 2025; Kapl et al., 2026), demonstrating benefits for reasoning, in-context learning, and test-time scaling (Geiping et al., 2025; Zhu et al., 2025; Saunshi et al., 2025; Yang et al., 2024; Alabdulmohsin & Zhai, 2025). Subsequent work explores relaxed weight sharing (Bae et al., 2025a), adaptive recursion depth (Bae et al., 2025b; Fu et al., 2025; Chen et al., 2025; Song et al., 2026; Elbayad et al., 2020; Jeddi et al., 2026; Goyal et al., 2026), dynamic skipping or looping (Li et al., 2026), and parallel test-time execution (Wu et al., 2025; Yang et al., 2026). Other studies investigate training stability (Prairie et al., 2026; Fu et al., 2026; Movahedi et al., 2026; Wang et al., 2026a), cross-loop memory and attention costs (Conchello Vendrell et al., 2026; O'Neill & Reid, 2026; Deng et al., 2026), MoE-compatible recurrence (Csordás et al., 2024; Lee et al., 2026; Chen et al., 2026b; Gao et al., 2026; Knupp et al., 2026; Chen et al., 2026c), and compute-matched scaling (Schwethelm et al., 2026; Prairie et al., 2026; Wang et al., 2026b). Despite these advances, pipeline-parallel training under cross-stage depth-recurrent feedback remains underexplored.

6 Conclusion

We identified a fundamental mismatch between conventional PP and looped Transformers: repeated cross-stage traversal introduces recurrent dependencies that independent stage-local repetition cannot preserve. We presented *Loop-PP*, to our knowledge the first pipeline-parallel training system for looped Transformers. Its loop-aware compiler represents logical recurrent visits independently of physical parameter storage and compiles them into dependency-aware Global Compute Orders and static Per-Rank Action Streams. Across 8B–60B models, Loop-PP reduces peak memory by up to 32.8% and improves the best measured throughput among the evaluated configurations by up to 12.7% under an equal resource budget. Our analysis further shows that its performance depends on recurrent placement and compute-chunk granularity, favoring layouts with sufficient independent local work.

AI USE STATEMENT

We used generative AI tools to assist with polishing the manuscript text, code navigation, plotting-script refinement, and consistency checks across the manuscript, experiment records, and implementation. The authors reviewed the resulting text, verified numerical claims against experiment records, inspected AI-assisted code changes, and checked cited works against their published or archived records. Generative AI was not treated as an experimental oracle or an independent source of evidence. The authors take responsibility for all claims, code, figures, and citations in this work.

ETHICS STATEMENT

This work studies distributed systems for model training and does not involve human subjects, personal data, or user studies. Its principal broader impact is the resource cost of large-scale accelerator experiments. We limit this cost by using short performance runs, reusing validated configurations, and reporting failed and out-of-memory runs rather than repeatedly searching them.

REPRODUCIBILITY STATEMENT

Appendix A gives the semantic construction and equivalence argument. Appendices B and C specify the scheduling and communication algorithms. Appendix G reports model shapes, parallel configurations, measurement rules, and the numerical data underlying the figures. The Supplementary Material provides the key implementation code.

REFERENCES

Thalaiyasingam Ajanthan, Sameera Ramasinghe, Yan Zuo, Gil Avraham, and Alexander Long. Nesterov method for asynchronous pipeline parallel optimization. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 909–924. PMLR, 2025. URL https://proceedings.mlr.press/v267/ajanthan25a.html.

Ibrahim Alabdulmohsin and Xiaohua Zhai. Recursive inference scaling: A winning path to scalable inference in language and multimodal systems. In *Advances in Neural Information Processing Systems*, 2025.

Sangmin Bae, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Seungyeon Kim, and Tal Schuster. Relaxed recursive transformers: Effective parameter sharing with layer-wise LoRA. In *International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=WwpYS0kkCt.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and Se-Young Yun. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. In *Advances in Neural Information Processing Systems*, 2025b. URL http://papers.nips.cc/paper_files/paper/2025/hash/8b08bbf8b420faa6eeb4020720582ec7-Abstract-Conference.html.

Yubing Bao, Zhihui Lu, Qiang Duan, Changze Lv, Xin Du, Zeyi Deng, Jingqi Feng, Sen Liu, Yang Chen, and Xin Wang. TPipe: Efficient spiking transformer training with time parallelism and asynchronous pipeline. In *IEEE INFOCOM 2026 - IEEE Conference on Computer Communications*. IEEE, 2026.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever,

and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Ling Chen, Houming Wu, and Wenjie Yu. AMDP: Asynchronous multi-directional pipeline parallelism for large-scale models training. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026a.

Wenkai Chen, Tianshu Li, Wenyong Huang, Yichun Yin, Lifeng Shang, and Chengwei Qin. LoopMoE: Unifying iterative computation with mixture-of-experts for language modeling. *arXiv preprint arXiv:2606.04438*, 2026b. doi: 10.48550/arXiv.2606.04438.

Yilong Chen, Junyuan Shang, Zhenyu Zhang, Yanxi Xie, Jiawei Sheng, Tingwen Liu, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. Inner thinking transformer: Leveraging dynamic depth scaling to foster adaptive internal thinking. *arXiv preprint arXiv:2502.13842*, 2025. doi: 10.48550/arXiv.2502.13842.

Yilong Chen, Naibin Gu, Junyuan Shang, Zhenyu Zhang, Yuchen Feng, Jiawei Sheng, Tingwen Liu, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. Mixture of universal experts: Scaling virtual width via depth-width transformation. *arXiv preprint arXiv:2603.04971*, 2026c. doi: 10.48550/arXiv.2603.04971.

Victor Conchello Vendrell, Arnau Padres Masdemont, Niccolò Grillo, Jordi Ros-Giralt, Arash Behboodi, and Fabio Valerio Massoli. Memory-efficient looped transformer: Decoupling compute from memory in looped language models. *arXiv preprint arXiv:2605.07721*, 2026. doi: 10.48550/arXiv.2605.07721.

Róbert Csordás, Kazuki Irie, Jürgen Schmidhuber, Christopher Potts, and Christopher D. Manning. MoEUT: Mixture-of-experts universal transformers. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/321387ba926b8e58d3591c0aeb52ffc2-Abstract-Conference.html.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

Chunyuan Deng, Yizhe Zhang, Rui-Jie Zhu, Yuanyuan Xu, Jiarui Liu, T. S. Eugene Ng, and Hanjie Chen. LT^2: Linear-time looped transformers. *arXiv preprint arXiv:2605.20670*, 2026. doi: 10.48550/arXiv.2605.20670.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJg7KhVKPH.

Rao Fu, Zixuan Yang, Jiankun Zhang, Jing Ma, Hechang Chen, Yu Li, and Yi Chang. Simply stabilizing the loop via fully looped transformer. *arXiv preprint arXiv:2605.18797*, 2026. doi: 10.48550/arXiv.2605.18797.

Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-hard: Selective latent iterations to improve reasoning language models. *arXiv preprint arXiv:2511.08577*, 2025. doi: 10.48550/arXiv.2511.08577. Accepted at ICML 2026.

Zitian Gao, Yilong Chen, Yihao Xiao, Xinyu Yang, Ran Tao, Joey Zhou, and Bryan Dai. Loop the loopies! *arXiv preprint arXiv:2607.16051*, 2026. doi: 10.48550/arXiv.2607.16051.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. In *Advances in Neural Information Processing Systems*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/3b01972cf31e6fa0fe29e4b8b5c2a0a1-Abstract-Conference.html.

Sahil Goyal, Swayam Agrawal, Gautham Govind Anil, Prateek Jain, Sujoy Paul, and Aditya Kusupati. ELT: Elastic looped transformers for visual generation. *arXiv preprint arXiv:2604.09168*, 2026. doi: 10.48550/arXiv.2604.09168.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022. doi: 10.48550/arXiv.2203.15556. URL https://papers.neurips.cc/paper_files/paper/2022/hash/c1e2faff6f588870935f114ebe04a3e5-Abstract-Conference.html.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32, pp. 103–112, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html.

Ahmadreza Jeddi, Marco Ciccone, and Babak Taati. LoopFormer: Elastic-depth looped transformers for latent reasoning via shortcut modulation. *arXiv preprint arXiv:2602.11451*, 2026. doi: 10.48550/arXiv.2602.11451.

Byungsoo Jeon, Mengdi Wu, Shiyi Cao, Sunghyun Kim, Sunghyun Park, Neeraj Aggarwal, Colin Unger, Daiyaan Arfeen, Peiyuan Liao, Xupeng Miao, Mohammad Alizadeh, Gregory R. Ganger, Tianqi Chen, and Zhihao Jia. GraphPipe: Improving performance and scalability of DNN training with graph pipeline parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 557–571. ACM, 2025. doi: 10.1145/3669940.3707220.

Ho Young Jhoo, Chung-Kil Hur, and Nuno P. Lopes. Pfeife: Automatic pipeline parallelism for PyTorch. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 27221–27236. PMLR, 2025. URL https://proceedings.mlr.press/v267/jhoo25a.html.

Hyunji Jung, Sungbin Shin, and Namhoon Lee. Mitigating staleness in asynchronous pipeline parallelism via basis rotation. In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2026.

Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962. doi: 10.1145/368996.369025.

Ferdinand Kapl, Emmanouil Angelis, Kaitlin Maile, Johannes von Oswald, and Stefan Bauer. From growing to looping: A unified view of iterative computation in LLMs. *arXiv preprint arXiv:2602.16490*, 2026. doi: 10.48550/arXiv.2602.16490.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. doi: 10.48550/arXiv.2001.08361.

Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. BPipe: Memory-balanced pipeline parallelism for training large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 16639–16653. PMLR, 2023. URL https://proceedings.mlr.press/v202/kim23l.html.

Jonas Knupp, Jan Hendrik Metzen, Jeremias Bohn, Georg Groh, and Kristian Kersting. Depth-recurrent attention mixtures: Giving latent reasoning the attention it deserves. *arXiv preprint arXiv:2601.21582*, 2026. doi: 10.48550/arXiv.2601.21582.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1eA7AEtvS.

Ryan Lee, Jacob Biloki, Edward J. Hu, and Jonathan May. Sparse layers are critical to scaling looped language models. *arXiv preprint arXiv:2605.09165*, 2026. doi: 10.48550/arXiv.2605.09165.

Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-level pipeline parallelism for training large-scale language models. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 6543–6552. PMLR, 2021. URL https://proceedings.mlr.press/v139/li21y.html.

Ziyue Li, Yang Li, and Tianyi Zhou. Skip a layer or loop it? learning program-of-layers in LLMs. *arXiv preprint arXiv:2606.06574*, 2026. doi: 10.48550/arXiv.2606.06574.

Sajad Movahedi, Vera Milovanović, Shlomo Libo Feigin, Alexander Theus, Thomas Hofmann, Valentina Boeva, T. Konstantin Rusch, and Antonio Orvieto. Fixed-point reasoners: Stable and adaptive deep looped transformers. *arXiv preprint arXiv:2606.18206*, 2026. doi: 10.48550/arXiv.2606.18206.

Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 7937–7947. PMLR, 2021a. URL https://proceedings.mlr.press/v139/narayanan21a.html.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–15, 2021b.

James O'Neill and Fergal Reid. Looped latent attention: Cross-loop KV compression for looped transformers. *arXiv preprint arXiv:2607.15456*, 2026. doi: 10.48550/arXiv.2607.15456.

Hayden Prairie, Zachary Novack, Taylor Berg-Kirkpatrick, and Daniel Y. Fu. Parcae: Scaling laws for stable looped language models. *arXiv preprint arXiv:2604.12946*, 2026. doi: 10.48550/arXiv.2604.12946.

Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. Pipeline parallelism with controllable memory. In *Advances in Neural Information Processing Systems*, 2024a. URL https://openreview.net/forum?id=Vvcnqs8091.

Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=tuzTN0eIO5.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE Press, 2020. doi: 10.1109/SC41405.2020.00024.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=din01GfZFd.

Kristian Schwethelm, Daniel Rueckert, and Georgios Kaissis. How much is one recurrence worth? iso-depth scaling laws for looped language models. *arXiv preprint arXiv:2604.21106*, 2026. doi: 10.48550/arXiv.2604.21106.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. doi: 10.48550/arXiv.1909.08053.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024. doi: 10.48550/arXiv.2408.03314.

Shixiang Song, He Li, Zitong Wang, Boyi Zeng, Feichen Song, Yixuan Wang, Zhiqin John Xu, Ziwei He, and Zhouhan Lin. AdaPonderLM: Gated pondering language models with token-wise adaptive depth. *arXiv preprint arXiv:2603.01914*, 2026. doi: 10.48550/arXiv.2603.01914.

Boran Sun, Guoyong Jiang, Lin Zhang, Chen Chen, Yuechen Tao, Zhishu Che, Jieling Yu, Shan Chang, Huaxi Gu, Fangming Liu, and Bo Li. Pulse: Training acceleration for large diffusion models with automatic pipeline parallelism. In *Proceedings of the 46th IEEE International Conference on Distributed Computing Systems*, pp. 360–370. IEEE, 2026.

Xinyi Wan, Penghui Qi, Guangxing Huang, Min Lin, and Jialin Li. PipeOffload: Improving scalability of pipeline parallelism with memory optimization. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 61942–61955. PMLR, 2025. URL https://proceedings.mlr.press/v267/wan25c.html.

Shaowen Wang, Bingrui Li, Ge Zhang, Wenhao Huang, Shen Yan, and Jian Li. On the residual scaling of looped transformers: Stability and transferability. *arXiv preprint arXiv:2606.18524*, 2026a. doi: 10.48550/arXiv.2606.18524.

Shaowen Wang, Ge Zhang, Kairong Luo, Yuhao Wu, Shaofan Liu, Jiaheng Liu, Wenhao Huang, Shen Yan, and Jian Li. SMELT: Scaling laws for compute-matched MoE looped transformers. *arXiv preprint arXiv:2609.01343*, 2026b. doi: 10.48550/arXiv.2609.01343.

Bohong Wu, Mengzhao Chen, Xiang Luo, Shen Yan, Qifan Yu, Fan Xia, Tianqi Zhang, Hongrui Zhan, Zheng Zhong, Xun Zhou, Siyuan Qiao, and Xingyan Bin. Parallel loop transformer for efficient test-time computation scaling. *arXiv preprint arXiv:2510.24824*, 2025. doi: 10.48550/arXiv.2510.24824.

Jian Yang, Shawn Guo, Wei Zhang, Tianyu Zheng, Yaxin Du, Haau-Sing Li, Jiajun Wu, Yue Song, Yan Xing, Qingsong Cai, Zelong Huang, Chuan Hao, Ran Tao, Xianglong Liu, Wayne Xin Zhao, Mingjie Tang, Weifeng Lv, Ming Zhou, and Bryan Dai. LoopCoder-v2: Only loop once for efficient test-time computation scaling. *arXiv preprint arXiv:2606.18023*, 2026. doi: 10.48550/arXiv.2606.18023.

Liu Yang, Kangwook Lee, Robert D. Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=HHbRxoDTxE.

Wenxuan Zhao, Yingfa Chen, Xu Han, Wenjing Han, Tianbo Huang, Zhiyu Li, Ao Sun, Jingheng Xu, Lin Gan, and Guangwen Yang. StateFlow: Sequence pipeline parallelism for long-context modeling with linear recurrence. *arXiv preprint arXiv:2608.06838*, 2026. doi: 10.48550/arXiv.2608.06838.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, Lu Li, Jiajun Shi, Kaijing Ma, Shanda Li, Taylor Kergan, Andrew Smith, Xingwei Qu, Mude Hui, Bohong Wu, Qiyang Min, Hongzhi Huang, Xun Zhou, Wei Ye, Jiaheng Liu, Jian Yang, Yunfeng Shi, Chenghua Lin, Enduo Zhao, Tianle Cai, Ge Zhang, Wenhao Huang, Yoshua Bengio, and Jason Eshraghian. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025. doi: 10.48550/arXiv.2510.25741.

Philip Zmushko, Egor Petrov, Nursultan Abdullaev, Mikhail Khrushchev, and Samuel Horváth. One-step gradient delay is not a barrier for large-scale asynchronous pipeline parallel LLM pretraining. *arXiv preprint arXiv:2606.30634*, 2026. doi: 10.48550/arXiv.2606.30634.

A Semantic Graph Construction Details

A.1 Formal Node and Edge Representation

For micro-batch m, let $\mathcal{G}_m = (\mathcal{V}_m, \mathcal{E}_m)$ denote its finitely unrolled forward graph. The Semantic Graph for M micro-batches is their disjoint union,

$$\mathcal{G} = \bigsqcup_{m=0}^{M-1} \mathcal{G}_m. \tag{11}$$

There are no model-semantic edges between different micro-batches; their interleaving is determined by the global compute scheduler.

A semantic node is represented as

$$v = \big(m_v,\, r_v,\, t_v,\, j_v,\, p_v,\, q_v,\, [\ell_v^{\text{start}}, \ell_v^{\text{end}})\big), \tag{12}$$

where m_v is the micro-batch, r_v is the computation role, t_v is the recurrent-step index, j_v is the segment index, (p_v, q_v) is the PP/VPP placement, and the final interval identifies the physical layer range when the node executes Transformer layers. The recurrent-step index is undefined for non-recurrent nodes, and the layer interval is empty for control-only nodes. The implementation distinguishes input, prelude, LTI precompute, recurrent entry, body, exit, coda, and output roles; fused entry/body/exit roles arise only after the optional graph rewrites described in Appendix D.

Let $\mu(v)$ denote the physical module invoked by node v. Logical visits at different recurrent steps are distinct nodes but may satisfy $\mu(v) = \mu(v')$. They therefore execute with the same parameter object and accumulate gradients into the same physical parameter storage.

A semantic edge is represented as

$$e = (u_e,\, v_e,\, \chi_e,\, \Sigma_e,\, \mathcal{B}_e)\,, \tag{13}$$

where u_e and v_e are the producer and consumer, χ_e is the logical channel, Σ_e is an ordered payload schema, and $\mathcal{B}_e$ maps each producer output slot to the corresponding consumer input slot. This record refines the body shorthand $e = (u, v, \Sigma_e)$; below, u and v abbreviate u_e and v_e when the channel and slot binding are not germane. Payload order is part of the communication contract. The principal channels are Main, Feedback, and Injection; representative payloads include the hidden activation, loop state, residual source, core input, injection source, and model-specific precomputed state. The schema associated with a backward transfer contains gradients for the same ordered payload fields.

A.2 Construction Procedure

The compiler constructs each $\mathcal{G}_m$ as follows.

1. It partitions the prelude, recurrent region, and coda into maximal segments with the same PP/VPP placement.
2. It creates the input and prelude nodes and connects consecutive prelude segments with main edges. The final prelude output becomes both the initial loop state and, when required by the model, a loop-invariant source.
3. For every recurrent step $t \in \{0, \ldots, \mathcal{T} - 1\}$, it instantiates the within-step recurrent DAG. At $t = 0$, the initial state enters through a main edge; at $t > 0$, the previous exit is connected to the current entry through a feedback edge.
4. It adds model-specific residual and injection edges, followed by the body segments and the recurrent transition at the exit.
5. It connects the final recurrent exit through the coda segments to the output node.

Every edge advances either the within-step topological order or the recurrent-step index. Consequently, the resulting graph is acyclic for finite $\mathcal{T}$. PP/VPP placement is an annotation on this graph: an edge whose endpoints share a PP stage remains a local dependency, whereas an edge crossing stages is later compiled into a planned transfer.

Algorithm 1 Semantic Graph Construction

Require: Model regions $\mathcal{P}, \mathcal{R}, \mathcal{C}$, recurrent-step count $\mathcal{T}$, micro-batch count M, PP/VPP placement

partition $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{C}$ into maximal same-placement segments
initialize $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$
for $m = 0, \ldots, M-1$ **do**
  $x \leftarrow \textsc{AddNode}(\mathcal{G}, m, \textsc{Input})$
  $a \leftarrow \textsc{AppendSegments}(\mathcal{G}, x, \mathcal{P}, m)$
  $x_{\text{prev}} \leftarrow \perp$
  for $t = 0, \ldots, \mathcal{T}-1$ **do**
    $(e_t, x_t, \mathcal{I}_t) \leftarrow \textsc{InstantiateRecurrent}(\mathcal{G}, \mathcal{R}, m, t)$
    if $t = 0$ **then**
      $\textsc{AddTypedEdge}(\mathcal{G}, a, e_t, \textsc{Main})$
    else
      $\textsc{AddTypedEdge}(\mathcal{G}, x_{\text{prev}}, e_t, \textsc{Feedback})$
    end if
    for all $v \in \mathcal{I}_t$ **do**
      $\textsc{AddTypedEdge}(\mathcal{G}, a, v, \textsc{Injection})$
    end for
    $x_{\text{prev}} \leftarrow x_t$
  end for
  $y \leftarrow \textsc{AppendSegments}(\mathcal{G}, x_{\text{prev}}, \mathcal{C}, m)$
  $\textsc{AddNodeAndEdge}(\mathcal{G}, y, m, \textsc{Output})$
end for
validate node identities, typed slot bindings, and acyclicity

return $\mathcal{G}$

Construction algorithm. Algorithm 1 makes the instantiation process explicit. APPENDSEGMENTS creates one node per segment visit and connects consecutive visits with typed main edges. INSTANTIATERECURRENT creates the placed nodes and within-step edges of one recurrent visit and returns its entry, exit, and designated injection consumers.

Example. Consider one micro-batch, two recurrent steps, and a recurrent region split across stages 0 and 1. Let A be the prelude output, E_t and X_t the entry and exit of step t, and R_t^0, R_t^1 the two placed recurrent segments. The main and feedback paths are

$$A \to E_0 \to R_0^0 \to R_0^1 \to X_0 \to E_1 \to R_1^0 \to R_1^1 \to X_1 \to C, \tag{14}$$

where $X_0 \to E_1$ is a feedback edge. If the prelude result is injected at both the entry and exit of each step, the graph additionally contains

$$A \to \{E_0, X_0, E_1, X_1\}. \tag{15}$$

These fan-out edges make the unrolled dataflow a non-chain DAG. If A, E_t, R_t^0 are placed on stage 0 and R_t^1, X_t on stage 1, the recurrent main, feedback, and exit-injection edges induce distinct cross-stage transfers with potentially different payload schemas. Although R_0^p and R_1^p are distinct logical nodes, they invoke the same physical segment on stage p.

A.3 DERIVED BACKWARD SEMANTICS

The Semantic Graph contains only forward model dependencies. Native-Guided Schedule Synthesis derives a forward operation (F, v) and a backward operation (B, v) for every $v \in \mathcal{V}$. For each forward edge $(u, v) \in \mathcal{E}$, it adds the forward dependency $(F, u) \to (F, v)$ and the reverse-mode dependency $(B, v) \to (B, u)$, together with $(F, v) \to (B, v)$.

For an output slot k of u, let $\mathrm{Out}(u, k)$ contain exactly the outgoing edges whose binding $\mathcal{B}_e$ consumes that slot. Its backward operation consumes the sum of the corresponding pullback contributions:

$$\nabla o_{u,k} = \sum_{e \in \mathrm{Out}(u,k)} \nabla_e o_{u,k}. \tag{16}$$

The runtime tracks this expected fan-in per producer slot and executes (B, u) only after all required contributions are available.

A.4 Semantic Equivalence

Theorem. Assume that each logical visit executes the same deterministic kernel given the same inputs, physical parameters, saved visit-local state, and logical random-number stream as its counterpart in a statically tied $\mathcal{T}$-step unrolling. During one training step, physical parameters remain fixed, and any other model-visible mutable state is either fixed or represented by semantic dependencies with the same update order. Assume also that every semantic edge preserves its ordered tensor bundle and slot binding exactly. Then any execution that respects the forward semantic dependencies and their derived reverse-mode dependencies produces the same model outputs and the same mathematical gradients for every input and shared physical parameter as that tied unrolling. Floating-point gradient values may differ only through permissible reordering of mathematically identical sums.

Proof. Order the forward nodes topologically. The initial input and prelude nodes agree with the tied unrolling by construction. Suppose all predecessors of node v produce the same output bundles as their counterparts in the tied unrolling. The schemas and slot bindings on the incoming semantic edges deliver exactly those bundles to the same physical kernel $\mu(v)$. The kernel therefore produces the same output, establishing forward equivalence by induction. In particular, the final coda output is identical.

For backward, traverse the derived reverse dependencies. The output gradient is identical because the model output is identical. For any node u and output slot k, Equation 16 accumulates exactly one contribution for every forward use selected by $\mathcal{B}_e$. Once these contributions match as a mathematical sum, applying the same local backward kernel to the same visit-local saved state yields the same input and physical-parameter gradient contributions. Reverse induction therefore establishes equality of every mathematical edge gradient. Summing the contributions from all logical visits gives the same mathematical gradient for each shared physical parameter; an implementation may accumulate those contributions in a different floating-point order. Replacing a local semantic edge with a communication transfer does not change this argument because the transfer preserves the schema and payload values by assumption. □

B Native-Guided Schedule Synthesis Details

B.1 Logical VPP Chunks

The native-table path first groups the semantic nodes of each $(\text{stage}, \text{micro-batch})$ pair into logical chunks. Every node that executes a nonempty physical layer range anchors one chunk. Control-only input, precompute, and entry nodes attach to the next layer-bearing anchor, whereas exit and output nodes attach to the preceding anchor. Nodes inside a backward chunk are visited in reverse order. This rule unfolds different recurrent steps into distinct logical chunks while retaining each node's physical VPP placement.

The implementation verifies that a stage has the same ordered chunk signature for every micro-batch. To remain consistent with Megatron's native interleaved-1F1B/VPP table, the native-table path retains its requirement that all PP stages have the same number N_c of logical chunks. Loop-PP also supports uneven logical-chunk counts across PP stages: in this case, the compiler falls back to the dependency-ready bounded-1F1B path rather than constructing an invalid native table. Thus, equal chunk counts are a requirement of the native-table path, not of Loop-PP as a whole.

B.2 Native Compute-Slot Tables

Let P be the number of PP stages and M the number of micro-batches. The native microbatch group size is the largest divisor of M no greater than $\min(P, M)$:

$$g = \max\{d : d \mid M,\ 1 \le d \le \min(P, M)\}. \tag{17}$$

Within each group of g micro-batches, the native table iterates over logical chunks first and micro-batches second. It therefore contains $N_{\text{slot}} = MN_c$ virtual-microbatch entries.

For PP stage p, the number of forward warmup entries is

$$w_p = \min\left(2(P-p-1) + (N_c-1)g,\ N_{\text{slot}}\right). \tag{18}$$

The implementation emits forward slots for table entries $0, \ldots, w_p - 1$. For each of the remaining $N_{\text{slot}} - w_p$ entries, it alternates one forward slot at index $w_p + i$ with one backward slot at index i. It then emits backward cooldown slots for entries $N_{\text{slot}} - w_p, \ldots, N_{\text{slot}} - 1$. For a forward slot, table chunk c selects logical chunk c; for a backward slot, it selects $N_c - c - 1$ and reverses the semantic nodes attached to that chunk. Thus the full forward/backward table on every stage has $J_p = 2N_{\text{slot}} = 2MN_c$ native compute slots, matching Equation 5. This procedure is the pure compiler equivalent of Megatron's interleaved-VPP compute table.

B.3 Semantic-Constrained Merge

The local native tables alone need not respect recurrent feedback or other semantic edges. The compiler therefore creates one key (ϕ, v) for every phase $\phi \in \{F, B\}$ and semantic node v. It adds the following constraints:

1. consecutive keys in every stage-local native table are ordered by a native edge;
2. each semantic edge (u, v) adds $(F, u) \to (F, v)$;
3. the same edge adds the reverse-mode constraint $(B, v) \to (B, u)$;
4. every node adds $(F, v) \to (B, v)$.

Each key also records its position $r_{\text{nat}}(\phi, v)$ in its owning stage's native table. The compiler topologically sorts the combined graph and selects ready keys lexicographically by native position, phase, and semantic-node identity. Compatibility between the native tables and the Semantic Graph is an invariant checked before scheduling. If the combined constraints contain a cycle, the native local tables conflict with the Semantic Graph and compilation fails rather than silently changing model semantics. Otherwise, the emitted order is the Global Compute Order used by communication planning.

Complete construction. Algorithm 2 records the native-table construction used by the main compiler. AppendSlot expands a virtual-table entry into the semantic nodes attached to its selected logical chunk and reverses their order for a backward slot. The heap key is exactly $(r_{\text{nat}}, r_{\text{phase}}, \text{id})$, with forward preceding backward only as a tie breaker after native position.

Pipeline bubbles. The generated stage-local slot tables have exactly the native warmup, alternating steady state, and cooldown structure for the same logical chunk table. Therefore, Loop-PP has the same structural pipeline bubble as native interleaved 1F1B.

C Communication-Aware Stream Compilation Details

C.1 Communication Planning and Cross-Stage Action DAG Construction

The compiler constructs one Communication Plan for each direction from the forward and backward subsequences of the native-guided Global Compute Order. Every remote semantic edge creates one planned transfer in each direction, represented by the five-tuple in Equation 8; every local edge remains a compute dependency. The logical channel χ_e is already part of e (Equation 13) and is not repeated in γ. Transfers are grouped by their consumer node y, where $(x, y) = (u, v)$ when $\phi = F$ and $(x, y) = (v, u)$ when $\phi = B$ for $e = (u, v, \Sigma_e)$. The executable plan then extends each five-tuple with FIFO and transport fields:

$$\gamma^+ = \left(e, \phi, s_{\text{src}}, s_{\text{dst}}, \Sigma_e^\phi,\ o_\gamma^{\text{ch}},\ o_\gamma^{\text{peer}},\ \kappa_\gamma\right), \qquad \text{base}(\gamma^+) = \gamma, \tag{19}$$

where o_γ^{ch} is contiguous within the key $(s_{\text{src}}, s_{\text{dst}}, \phi, \chi_e)$, o_γ^{peer} is contiguous within the directed-peer key $(s_{\text{src}}, s_{\text{dst}}, \phi, \kappa_\gamma)$, and κ_γ selects adjacent-stage or direct transport. For readability, the Action-DAG notation continues to identify a transfer by its semantic projection γ; the corresponding executable record γ^+ supplies the FIFO and transport attributes used when constructing and emitting those actions.

Algorithm 2 Native-Guided Schedule Synthesis

Require: Semantic graph $\mathcal{G}$ with P stages and M micro-batches

$\mathcal{X} \leftarrow$ LOGICALCHUNKS($\mathcal{G}$); verify signatures across micro-batches within each stage and a common chunk count N_c across stages
$g \leftarrow \max\{d : d \mid M,\ 1 \le d \le \min(P, M)\}$
$\mathcal{U} \leftarrow [\,]$
for $m_0 = 0, g, \ldots, M - g$ **do**
  for $c = 0, \ldots, N_c - 1$ **do**
    for $m = m_0, \ldots, m_0 + g - 1$ **do**
      append (m, c) to $\mathcal{U}$
    end for
  end for
end for
for $p = 0, \ldots, P - 1$ **do**
  $w_p \leftarrow \min(2(P - p - 1) + (N_c - 1)g,\ |\mathcal{U}|)$
  $\mathcal{N}_p \leftarrow [\,]$
  for $i = 0, \ldots, w_p - 1$ **do**
    APPENDSLOT($\mathcal{N}_p, F, \mathcal{U}[i], \mathcal{X}[p]$)
  end for
  for $i = 0, \ldots, |\mathcal{U}| - w_p - 1$ **do**
    APPENDSLOT($\mathcal{N}_p, F, \mathcal{U}[w_p + i], \mathcal{X}[p]$)
    APPENDSLOT($\mathcal{N}_p, B, \mathcal{U}[i], \mathcal{X}[p]$)
  end for
  for $i = |\mathcal{U}| - w_p, \ldots, |\mathcal{U}| - 1$ **do**
    APPENDSLOT($\mathcal{N}_p, B, \mathcal{U}[i], \mathcal{X}[p]$)
  end for
end for
$\mathcal{K} \leftarrow \{(F, v), (B, v) : v \in \mathcal{V}\}$; initialize constraint graph $\mathcal{H}$ on $\mathcal{K}$
for all consecutive keys a, b in each expanded $\mathcal{N}_p$ **do**
  ADDEDGE($\mathcal{H}, a, b$); record each key's position in the expanded sequence as r_{nat}
end for
for all $v \in \mathcal{V}$ **do**
  ADDEDGE($\mathcal{H}, (F, v), (B, v)$)
end for
for all $(u, v) \in \mathcal{E}$ **do**
  ADDEDGE($\mathcal{H}, (F, u), (F, v)$); ADDEDGE($\mathcal{H}, (B, v), (B, u)$)
end for
$Q \leftarrow$ all zero-indegree keys in $\mathcal{H}$, keyed by $(r_{\mathrm{nat}}, r_{\mathrm{phase}}, \mathrm{id})$
$\mathcal{O} \leftarrow [\,]$
while $Q \neq \varnothing$ **do**
  $k \leftarrow$ POPMIN(Q); append k to $\mathcal{O}$
  for all $z \in \mathrm{Succ}_{\mathcal{H}}(k)$ **do**
    decrement $\mathrm{indegree}(z)$; insert z into Q when it reaches zero
  end for
end while
if $|\mathcal{O}| \neq |\mathcal{K}|$ **then**
  error: native and semantic constraints are cyclic
end if

return $\mathcal{O}$

The compiler creates one integrated COMPUTE action for every phased semantic operation. It additionally creates one ENSUREBATCH action per (stage, micro-batch) pair and, for both endpoints of every planned transfer, one POST and one WAIT. In the implementation these paper-level action names correspond to `ENSURE_MICROBATCH`, `TRANSFER_POST`, and `TRANSFER_WAIT`. At the sender, WAIT protects the source buffer from premature reuse and releases the outstanding-send

slot on completion; at the receiver, it makes the payload available to consumers and releases the receive slot. The Cross-Stage Action DAG then adds:

1. forward semantic edges and their reversed backward counterparts;
2. each node's forward-before-backward edge;
3. ENSUREBATCH before every forward compute and transfer of that micro-batch;
4. producer compute before the sender POST;
5. both matching endpoint POSTs before either endpoint WAIT;
6. receiver WAIT before consumer compute;
7. contiguous POST and WAIT order for each channel and ordered peer;
8. ordering barriers for the first dynamic metadata transfer of a schema;
9. the previous receive WAIT before the next receive POST on a stage when the static-shape fast path is used.

The final rule enforces the compiler's single outstanding receive window. It is necessary because multiple unmatched receives on the shared NCCL communicator can block local compute that is required to produce their matching sends. The resulting DAG is validated for acyclicity before linearization.

C.2 NATIVE-AWARE COMMUNICATION LINEARIZATION

Loop-PP uses a two-level static compiler. It first traverses the Cross-Stage Action DAG to form a global sequence of communication *epochs*. It then projects these epochs onto the immutable native compute timeline of each rank. This separation gives both endpoints the same communication matching decisions while allowing each rank to reach an epoch after its own local producer. No global ready-queue decision is made at runtime.

Global communication epochs. A transfer becomes structurally ready only when the POST actions at both endpoints have zero remaining predecessors. Two ready transfers may form one bidirectional epoch when their source and destination stages are reversed. The compiler excludes the first transfer of an uncached payload schema from pairing because that transfer must establish dynamic metadata before the static-shape fast path can be used. Within a paired epoch, the lower-numbered stage emits its send before its receive, while the higher-numbered stage emits its receive before its send, matching the native P2P handshake.

The epoch compiler maintains one send slot and one receive slot per stage. At each frontier, it prefers a feasible bidirectional epoch, then an unpaired first-metadata epoch, and then the next dependency-ready compute on the immutable native timeline. If these choices are blocked, it emits only the WAITs needed to release a required slot, satisfy a metadata or FIFO barrier, or unblock the next native compute. A transfer is emitted alone only when no reverse-direction partner can be used at the current frontier. Among legal pairs, the implementation prefers one whose consumers lie ahead of the current compute cursors and then favors opposite-phase communication; the communication-plan sequence numbers provide deterministic tie breaking.

Algorithm 3 records this priority order. The compiler updates Action-DAG predecessor counts after every emitted action; consequently, each helper returns only dependency-ready actions. PAIR excludes first-metadata transfers and requires reversed endpoints, while BLOCKERWAIT returns the ready WAITs that occupy a send or receive slot needed by a candidate epoch.

Per-rank projection. Once the global epochs are fixed, the compiler projects them independently onto each stage. Let $\mathcal{N}_s^{\mathrm{nat}}$ denote stage s's native compute sequence and $\mathcal{Z}_s$ the subsequence of communication epochs owned by s. The projection repeatedly emits the next locally ready epoch when the corresponding send and receive slots are free; otherwise it emits the next locally ready operation in $\mathcal{N}_s^{\mathrm{nat}}$. This is the key overlap step: a receive may remain open while computation from another logical VPP chunk executes. If neither can advance, the compiler retires the send or receive that blocks the next epoch or the receive WAIT directly required by the next compute. Control actions are placed whenever they become the remaining locally ready work.

Algorithm 3 Native Communication-Epoch Compilation

Require: Action DAG $\mathcal{A}$, plans Π_F, Π_B, native compute timelines $\{\mathcal{N}_s^{\mathrm{nat}}\}$
initialize ready sets, compute cursors, $send_s = recv_s = \bot$, and epoch list $\mathcal{Z} \leftarrow [\,]$
while some Action-DAG vertex is not emitted **do**
  if a protocol-required unpaired receive has a ready WAIT **then**
    emit its same-stage send WAIT first when ready; otherwise emit the receive WAIT; **continue**
  end if
  $R \leftarrow$ transfers whose two endpoint POSTs are ready, in communication-plan order
  $Z \leftarrow$ highest-priority capacity-feasible element of $\textsc{Pair}(R)$
  if $Z \neq \bot$ **then**
    emit both endpoint POSTs of Z; append Z to $\mathcal{Z}$; **continue**
  end if
  $\gamma \leftarrow$ first capacity-feasible first-metadata transfer in R
  if $\gamma \neq \bot$ **then**
    emit both POSTs of γ; append (γ) to $\mathcal{Z}$; **continue**
  end if
  $a \leftarrow$ minimum-priority dependency-ready next action among the cursors of $\{\mathcal{N}_s^{\mathrm{nat}}\}$
  if $a \neq \bot$ **then**
    emit a and advance its stage cursor; **continue**
  end if
  if a ready micro-batch control action exists **then**
    emit its minimum-identity action; **continue**
  end if
  $Z \leftarrow$ highest-priority pair in $\textsc{Pair}(R)$, or the first singleton in R
  if a ready $\textsc{BlockerWait}(Z)$ exists **then**
    emit its minimum-identity WAIT; **continue**
  end if
  if a ready WAIT directly precedes a stage's next native compute **then**
    emit a ready same-stage send WAIT first when present; otherwise emit that predecessor; **continue**
  end if
  if any WAIT is ready **then**
    emit its minimum-identity action; **continue**
  end if
  $\gamma \leftarrow$ first capacity-feasible singleton in R
  if $\gamma \neq \bot$ **then**
    emit both POSTs of γ; append (γ) to $\mathcal{Z}$; **continue**
  end if
  error: no globally legal action
end while

return $\mathcal{Z}$

Algorithm 4 gives the product-path projection. READY tests only predecessors owned by the same stage, because cross-rank matching has already been fixed by the global epochs. OPEN and CLOSE update the single outstanding send and receive slots. The branch for a separately deferred wgrad is omitted because the evaluated configuration uses integrated backward compute.

When only one local virtual chunk is available, there is no second local chunk with which to fill the receive window. The implementation therefore applies a separate static repair to the same compiled epoch order: when locally legal, it promotes an older backward send (and, when required, its receive completion) before the next forward receive completion. The repair removes pair annotations and is accepted only if all stage-local DAG edges, FIFO constraints, and the one-send/one-receive bounds remain valid. When multiple local chunks are available, the paired projection above is used directly.

Finally, after communication packing and any static repair, the compiler validates complete action coverage, every stage-local Action-DAG edge, per-edge FIFO order, mirrored endpoint pairing, and

Algorithm 4 Native-Aware Per-Rank Action Linearization

Require: Action DAG $\mathcal{A}$, native compute sequences $\{\mathcal{N}_s^{\text{nat}}\}$, globally compiled epochs $\mathcal{Z}$

for $s = 0, \ldots, P-1$ **do**
  $\mathcal{Q}_s \leftarrow [\,]$; $i_C \leftarrow 0$; $i_Z \leftarrow 0$; $send \leftarrow \bot$; $recv \leftarrow \bot$
  $\mathcal{Z}_s \leftarrow$ projection of $\mathcal{Z}$ onto endpoint actions owned by s
  while some action owned by s is not emitted **do**
    $Z \leftarrow \mathcal{Z}_s[i_Z]$ if $i_Z < |\mathcal{Z}_s|$, otherwise $\bot$
    if $Z \neq \bot \wedge \textsc{Ready}(Z) \wedge \textsc{SlotsFree}(Z, send, recv)$ **then**
      append the parity-ordered POST action(s) of Z to $\mathcal{Q}_s$
      $\textsc{Open}(Z, send, recv)$; $i_Z \leftarrow i_Z + 1$; **continue**
    end if
    $a \leftarrow \mathcal{N}_s^{\text{nat}}[i_C]$ if $i_C < |\mathcal{N}_s^{\text{nat}}|$, otherwise $\bot$
    if $a \neq \bot \wedge \textsc{Ready}(a)$ **then**
      append a to $\mathcal{Q}_s$; $i_C \leftarrow i_C + 1$; **continue**
    end if
    if a ready micro-batch control action exists **then**
      append the minimum-identity ready control action to $\mathcal{Q}_s$; **continue**
    end if
    if $Z \neq \bot \wedge \textsc{Ready}(Z)$ but an outstanding slot blocks Z **then**
      close a ready send first, otherwise close a ready receive; **continue**
    end if
    if $a \neq \bot$ has a ready receive-WAIT predecessor **then**
      close a ready send first if required by the native boundary
      otherwise close that receive; **continue**
    end if
    if any outstanding transfer has a ready WAIT **then**
      close its minimum-identity ready WAIT; **continue**
    end if
    error: no locally legal action
  end while
  verify that every epoch was projected and no transfer remains open
end for
after runtime communication packing, validate action coverage, local DAG order, communication FIFO, mirrored pairing, the send/receive caps, and progress-graph acyclicity

return $\{\mathcal{Q}_s\}_{s=0}^{P-1}$

the one-send/one-receive invariant. It then constructs the augmented progress graph from the actual runtime action units and rejects the compilation if that graph contains a cycle. The resulting streams are immutable *Per-Rank Action Streams*; execution only interprets these precompiled actions.

C.3 Correctness of Compiled Streams

Planned-transfer semantics. The interpreter processes each per-rank stream in order: a COMPUTE action completes before the interpreter advances, a POST advances only after every operation in its actual runtime action unit has been successfully submitted to a backend execution context without waiting for its peer, and a WAIT advances only after completion. Backend ordering dependencies that precede progress eligibility of a submitted operation are included in the augmented progress graph defined below. For every planned transfer, the sender POST binds the payload produced by its source compute, and the source buffer remains valid until the sender WAIT completes. A transfer can complete only after its endpoint operations have been issued. The destination WAIT returns only after that transfer has completed and its ordered payload is available to the consumer. Transport preserves the payload schema and values.

For communication matching, let $d(\gamma)$ denote the backend matching domain of transfer γ, including its communicator, directed peer, and any transport-specific tag namespace. Operations in distinct domains cannot match. No unplanned P2P operation shares a domain with the compiled transfers

during their execution. Within one domain, transfer identities are unique, and the backend either matches that identity explicitly or, as in the evaluated ordered-P2P path, matches sends and receives in issue order. The latter path therefore requires both endpoints to issue the common epoch-induced subsequence for that domain.

Proposition 1 (Semantic-order preservation). Suppose compilation succeeds and the generated streams $\{\mathcal{Q}_s\}_{s=0}^{P-1}$ pass the validations above. Then any completed joint execution under the planned-transfer semantics respects every forward and reverse-mode dependency of the Semantic Graph, as well as forward-before-backward order for every semantic node.

Proof. Let $\prec_{\text{exec}}$ denote happens-before in the completed joint execution. Action coverage gives exactly one forward and one backward compute action for every semantic node and the required endpoint actions for every remote transfer. Because every $\mathcal{Q}_s$ is validated as a linear extension of the Action DAG induced on stage s, each stage-local DAG edge is preserved by stream order. The interpreter semantics then turns such stream order into execution order: an earlier blocking action completes, or an earlier POST is issued, before the interpreter starts the later action.

Consider an arbitrary semantic edge $e = (u, v, \Sigma_e)$. If e is local, its endpoints occupy the same stage. The Action DAG contains

$$\text{Compute}^F(u) \prec \text{Compute}^F(v) \quad \text{and} \quad \text{Compute}^B(v) \prec \text{Compute}^B(u).$$

Both are stage-local edges and hence hold under $\prec_{\text{exec}}$ by the validated linear-extension property.

If e is remote, fix a phase ϕ and its planned transfer γ. Write $(x, y) = (u, v)$ for $\phi = F$ and $(x, y) = (v, u)$ for $\phi = B$. Equation 9 places the source compute before the sender POST on the source stage and the destination WAIT before the consumer compute on the destination stage. Stage-local validation and ordered interpretation therefore give

$$\text{Compute}^\phi(x) \text{ completes } \prec_{\text{exec}} \text{POST}_{\text{src}}(\gamma) \text{ is issued}$$

and

$$\text{WAIT}_{\text{dst}}(\gamma) \text{ returns } \prec_{\text{exec}} \text{Compute}^\phi(y) \text{ starts.}$$

By planned-transfer causality, the transfer cannot complete before its sender POST is issued, and its destination WAIT cannot return before completion. Combining these relations yields

$$\text{Compute}^\phi(x) \text{ completes } \prec_{\text{exec}} \text{POST}_{\text{src}}(\gamma) \text{ is issued } \prec_{\text{exec}} \text{transfer}(\gamma) \text{ completes } \prec_{\text{exec}} \text{WAIT}_{\text{dst}}(\gamma) \text{ returns } \prec_{\text{exec}} \text{Compu}$$

Thus the forward dependency is preserved when $\phi = F$, and the derived reverse-mode dependency is preserved when $\phi = B$.

Finally, for every semantic node v, the Action DAG contains the stage-local edge $\text{Compute}^F(v) \prec \text{Compute}^B(v)$, which is preserved by the same argument. The local and remote cases exhaust all semantic edges, so every claimed dependency holds. □

Proposition 2 (Communication matching safety). Suppose compilation succeeds, the generated streams pass the action-coverage, FIFO, and mirrored-endpoint validations above, and each backend matching domain obeys the matching semantics stated above. Then every send has exactly one intended receive with the same transfer identity and payload schema, and no send can cross-match with a different planned transfer.

Proof. Every remote semantic edge and phase creates one executable transfer record γ^+ with identity determined by the unique pair (e, ϕ). Its identity and payload schema are shared by the sender and receiver endpoint records. Mirrored-endpoint validation requires one sender POST and one destination POST for that same record, while complete action coverage rejects a missing endpoint, a duplicate endpoint, or an endpoint that does not belong to a planned transfer.

Fix an arbitrary backend matching domain d. Filter the global epoch order to the transfers in that domain and denote the resulting sequence by

$$L_d = [\gamma_0, \gamma_1, \ldots, \gamma_{q-1}].$$

The epoch compiler inserts both endpoint POSTs of every γ_i at the same global epoch position. The projection in Algorithm 4 advances its epoch cursor monotonically. It may interleave local compute and WAIT actions, but cannot skip or permute the epoch subsequence owned by either endpoint. The final FIFO and mirrored-endpoint validations consequently establish that the sender issues

$$[\mathrm{send}(\gamma_0), \ldots, \mathrm{send}(\gamma_{q-1})]$$

and the receiver issues

$$[\mathrm{recv}(\gamma_0), \ldots, \mathrm{recv}(\gamma_{q-1})]$$

in exactly the same order, with neither omissions nor duplicates.

If the backend uses explicit transfer identities in d, uniqueness and equal endpoint identities force $\mathrm{send}(\gamma_i)$ to match $\mathrm{recv}(\gamma_i)$. Otherwise the backend uses FIFO matching. For $i = 0$, the first unmatched operations at the two endpoints are the send and receive of γ_0, so they match. Assume inductively that the first i pairs have matched as intended. The first unmatched operations are then the send and receive of γ_i; FIFO matching pairs them, and they cannot match any γ_j with $j \neq i$. Induction proves intended matching for all q transfers in d. Since distinct matching domains cannot cross-match, applying the argument independently to every d proves the global one-send/one-receive and no-cross-match properties. Finally, the paired endpoints are projections of the same γ^+, so their transfer identity and ordered payload schema agree. The logical-channel FIFO validation further preserves the channel-specific order required by the consumer. □

Augmented progress graph. Here an action is an actual runtime action unit after communication packing, native-table projection, and any static repair; operations submitted atomically by one packed POST share one POST milestone. For every such action a in a final stream, let $\widehat{a}$ denote the milestone at which its rank may advance: completion for COMPUTE and control actions, successful issue for POST, and return for WAIT. The augmented progress graph $\mathcal{G}_{\mathrm{prog}}$ contains one such milestone for every action. For each consecutive pair $a_{s,i}, a_{s,i+1}$ in $\mathcal{Q}_s$, it contains the local stream edge

$$\widehat{a}_{s,i} \longrightarrow \widehat{a}_{s,i+1}.$$

For every planned transfer γ, it additionally contains

$$\widehat{\mathrm{POST}}_{\mathrm{src}}(\gamma), \widehat{\mathrm{POST}}_{\mathrm{dst}}(\gamma) \longrightarrow \widehat{\mathrm{WAIT}}_{\mathrm{src}}(\gamma),$$
$$\widehat{\mathrm{POST}}_{\mathrm{src}}(\gamma), \widehat{\mathrm{POST}}_{\mathrm{dst}}(\gamma) \longrightarrow \widehat{\mathrm{WAIT}}_{\mathrm{dst}}(\gamma).$$

These edges model the fact that either endpoint completion may require both endpoint operations to have been issued. Importantly, the local edges are derived from the *final* runtime streams, after all transformations that can coalesce or reorder operations. The compiler checks $\mathcal{G}_{\mathrm{prog}}$ for acyclicity in $O(|V| + |E|)$ time before execution.

Proposition 3 (Conditional PP-transport deadlock freedom). Suppose the generated streams pass the validations above and $\mathcal{G}_{\mathrm{prog}}$ is acyclic. Assume a fair execution in which (i) every enabled compute or control action terminates in finite time, (ii) a POST submits its packed operations and returns in finite time without waiting for its peer, (iii) once both endpoint POSTs of a planned transfer have become progress-eligible, the transport completes that transfer in finite time even if an endpoint interpreter is waiting, and (iv) the compiled PP runtime introduces no additional blocking synchronization outside $\mathcal{G}_{\mathrm{prog}}$. Then the compiled PP action streams contain no circular wait and complete in finite time; this claim does not extend to failures or synchronization outside the compiled PP runtime.

Proof. Assume for contradiction that a fair execution reaches a permanent state with at least one unfinished milestone. The subgraph of $\mathcal{G}_{\mathrm{prog}}$ induced by the unfinished milestones is a nonempty finite DAG, so it has a vertex $\widehat{a}$ with no unfinished predecessor. Hence every predecessor of $\widehat{a}$ has completed. Its local-stream predecessor, if any, has therefore completed, so a is the next action of its rank.

If a is a compute or control action, fairness and assumption (i) imply that it completes. If a is a POST, fairness and assumption (ii) imply that it is issued. If a is a WAIT, both endpoint POST milestones are predecessors of $\widehat{a}$ and have completed, including their modeled backend-ordering predecessors; assumptions (iii) and fairness therefore imply that the transfer completes and the WAIT returns. In

every case an unfinished milestone advances, contradicting permanence of the assumed state. Thus no permanent blocked state with unfinished actions is reachable. Because the streams are finite, repeated progress completes all of them. □

Consequence and scope. Proposition 1 establishes semantic dependency order, and Proposition 2 discharges the transport-matching premise for the actual endpoint operations. Together with the assumptions of the Semantic Equivalence theorem in Appendix A, they establish forward and backward semantic preservation for every completed joint execution. Proposition 3 separately establishes PP-transport deadlock freedom under the stated fairness and transport-progress assumptions. The compiler validates the structural premise—acyclicity of the augmented progress graph over the final runtime action units—but cannot validate backend progress, hardware failure freedom, or synchronization introduced outside the compiled PP runtime.

D Implementation Details

D.1 Semantic Supernodes and Local Side Inputs

The graph builder initially gives recurrent entry, layer-bearing body, and transition exit separate semantic identities. This representation is convenient for validating slot bindings and different transition functions, but a stage-local entry or exit otherwise creates an extra tensor handoff, detach leaf, autograd invocation, and runtime action. With semantic supernodes enabled, the builder replaces

$$\textsc{Entry} \rightarrow \textsc{Body}_0 \quad \text{and} \quad \textsc{Body}_{J-1} \rightarrow \textsc{Exit}$$

by EntryBody and BodyExit, respectively; a one-segment recurrent region becomes one EntryBodyExit node. The fused node retains the union of the original input/output slots and executes the same entry transition, Transformer layers, and exit transition in that order. This is a semantic-graph rewrite rather than scheduler-only fusion: the removed local edges and their backward counterparts never enter the Action DAG. This optional rewrite applies when the recurrent region has identifiable entry and exit transitions around one or more layer-bearing body segments, and when fusion preserves all input/output slot bindings. Accordingly, the rewrite is applied only when entry/exit injection edges and any locally routed CoreInput do not require the affected nodes to remain separately addressable.

Recurrent transitions may consume a residual source and, for delta updates, the input to the recurrent core. The generic schema can relay these side tensors through every body edge. The localized path instead adds a direct stage-local edge when the source and exit transition share a stage. In a full-loop placement, the initial residual may still need to traverse the core when the prelude seed and exit are on different stages; later visits use the preceding exit as their source and can keep that tensor local. The graph builder then removes the localized field from intermediate core bundle schemas. This shortens payload lifetime and reduces both transmitted bytes and tensor bookkeeping without changing the hidden-state or feedback dependencies.

D.2 Planned Bundle Transport

Every planned transfer carries a statically ordered payload schema. The transport key consists of source and destination stages, direction, semantic channel, schema identity, and ordered payload fields. On the first transfer for a key, the sender constructs a compact header containing tensor shapes and dtypes and the receiver allocates the corresponding bundle. When static-shape communication is enabled, this metadata is retained by the training schedule across optimizer steps. Later transfers validate that the sender metadata has not changed, allocate directly from the cached description, and omit the header. A first-metadata transfer is emitted as an unpaired epoch so that both endpoints establish the cache before the native paired fast path is used.

POST invokes nonblocking P2P communication and WAIT completes it. The transport uses CUDA-stream waits so completion introduces a stream dependency rather than a CPU-side device synchronization. The epoch compiler pairs reversed source/destination transfers only when both endpoint POSTs are dependency-ready and their metadata is already known. It follows the native parity rule: the lower-numbered stage posts send before receive, and the higher-numbered stage

posts receive before send. Main traffic between adjacent stages may use the existing adjacent transport; feedback, injection, and non-adjacent edges use their planned direct peers. The implementation admits at most one open send and one open receive per stage on the shared NCCL communicator. This still permits bidirectional overlap while preventing a sequence of unmatched receives from blocking local producer computation.

D.3 Compilation Caches and Validation

The recurrent executor object is constructed for each training step, but the owning Megatron schedule retains two run-level caches. The communication metadata cache described above is indexed by payload protocol. The compiled-schedule cache is indexed by the semantic-graph hash, PP stage, scheduling policy, transport and Action-DAG options, expert-parallel configuration, and the set of metadata schemas already established. Its value contains the phased compute order, forward and backward communication plans, per-rank action streams, preflight signature, and backward-tape specification. Consequently, a cache hit does not rebuild the global constraints or re-run Action-DAG linearization.

We quantify this benefit with a representative PP4 execution trace. Across ranks, a compiled-schedule cache miss incurs a median recurrent-executor setup time of $1.68\,\mathrm{s}$, including $1.00\,\mathrm{s}$ for executor construction and $0.66\,\mathrm{s}$ for distributed preflight. After the schedule is cached, the median setup time falls to $31.9\,\mathrm{ms}$ ($52.7\times$ lower), with construction and preflight reduced to $28.7\,\mathrm{ms}$ and $0.7\,\mathrm{ms}$, respectively. Because the executor object is instantiated at every training step, caching turns this repeated global compilation and validation cost into a small local lookup and stream-setup cost.

Compilation is accepted only after checking semantic and Action-DAG acyclicity, complete forward/backward node coverage, payload slot bindings, channel and directed-peer sequence numbers, stage-local dependency order, mirrored P2P pair annotations, and the outstanding-transfer bounds. Before execution, ranks exchange compact preflight signatures derived from the graph, plans, and streams; a mismatch fails before issuing training traffic. At runtime, each stage advances through its immutable Action-DAG stream.

D.4 Backward and MoE Boundaries

Loop-PP invokes one physical module from multiple logical visits, so standard autograd accumulation produces one physical parameter gradient after all semantic fan-in contributions arrive. The evaluated product configuration uses integrated backward computation on all pipeline stages: weight-gradient work is contained in the corresponding backward action rather than exposed as a separately scheduled ComputeW vertex. This avoids retaining multiple deferred weight-gradient states for a shared Transformer-Engine layer.

The pipeline compiler treats a Transformer layer, including its MoE router, dispatch, expert kernels, and combine collective, as one local compute operation. It may overlap pipeline communication with another dependency-ready layer, but it does not reorder collectives inside a layer. This boundary is why expert all-to-all time is a nested observation inside T_{local} rather than a separate additive pipeline phase.

E Performance Analysis

E.1 Bubble Analysis

We follow the bubble accounting of PipeOffload (Wan et al., 2025). Let P be the number of pipeline devices, M the number of micro-batches, and N_c the number of balanced logical chunks placed on each device; these correspond to d, m, and v, respectively, in PipeOffload. Let

$$T_{\mathrm{chunk}} = T_F + T_B + T_W \tag{20}$$

be the forward, input-gradient, and weight-gradient time of one logical chunk. When backward computation is not explicitly split, $T_B + T_W$ is simply the total backward-action time.

Under balanced chunk costs, conventional non-interleaved 1F1B and native interleaved 1F1B have bubble times

$$T_{\mathrm{bub}}^{\mathrm{1F1B}} = N_c(P-1)T_{\mathrm{chunk}}, \qquad T_{\mathrm{bub}}^{\mathrm{int}} = (P-1)T_{\mathrm{chunk}}. \tag{21}$$

To see this, measure time in units of one balanced chunk execution. Non-interleaved 1F1B fills and drains across $P-1$ stage boundaries, and each interval contains all N_c chunks assigned to a device, giving $N_c(P-1)$ idle chunk-times. Interleaving staggers these chunks across devices, so the same fill and drain contains only $P-1$ idle chunk-times. The useful per-device compute time is MN_cT_{chunk}, giving

$$\rho_{\text{bub}}^{\text{1F1B}} = \frac{P-1}{M+P-1}, \qquad \rho_{\text{bub}}^{\text{int}} = \frac{P-1}{MN_c+P-1}. \tag{22}$$

Loop-PP uses the same stage-local interleaved slot table for the same (P, M, N_c). Its structural bubble is therefore exactly that of native interleaved 1F1B:

$$T_{\text{bub}}^{\text{Loop-PP}} = T_{\text{bub}}^{\text{int}}, \qquad \rho_{\text{bub}}^{\text{Loop-PP}} = \rho_{\text{bub}}^{\text{int}} = \frac{P-1}{MN_c+P-1}. \tag{23}$$

Thus, Loop-PP does not claim a bubble reduction over interleaved 1F1B with the same logical chunk table.

For the $P=2$, $M=2$, $N_c=2$ example in our method figure, both Loop-PP and interleaved 1F1B have

$$\rho_{\text{bub}} = \frac{1}{2\cdot 2+1} = 20\%. \tag{24}$$

At the same (P, M), non-interleaved 1F1B has a bubble ratio of $1/(2+1) = 33.3\%$.

E.2 Recurrent Communication and Exposed Pipeline Overhead

The equal structural bubble above does not imply equal communication cost. We compare Loop-PP with the Unrolled baseline, which executes the equivalent effective model depth using native interleaved 1F1B. For this analysis, we use the standard cyclic interleaved placement with P pipeline devices, V virtual stages per device, M micro-batches, and $\mathcal{T}$ recurrent steps. This placement provides a common analytical setting for the two schedules; it is not a system restriction, and Loop-PP also supports other valid PP/VPP placements represented by the Semantic Graph. Unrolled forms a linear pipeline of PV virtual stages, so its forward and backward passes together perform

$$N_{\text{transfer}}^{\text{Unrolled}} = 2M(PV-1) \tag{25}$$

logical cross-stage transfers.

In Loop-PP, each recurrent step traverses the same PV virtual stages. Each step contains $PV-1$ within-step transitions, and every pair of consecutive recurrent steps adds one feedback transition from the final stage of the previous step to the first stage of the next. The corresponding transfer count is therefore

$$\begin{aligned} N_{\text{transfer}}^{\text{Loop-PP}} &= 2M\left[\mathcal{T}(PV-1)+(\mathcal{T}-1)\right] \\ &= 2M(\mathcal{T}PV-1). \end{aligned} \tag{26}$$

Thus, when both methods use the same VPP degree, repeated traversal adds

$$N_{\text{transfer}}^{\text{Loop-PP}} - N_{\text{transfer}}^{\text{Unrolled}} = 2MPV(\mathcal{T}-1) \tag{27}$$

logical transfers. If the two methods use different VPP degrees, Equations equation 25 and equation 26 apply with their respective values of V.

Transfer count alone does not determine communication volume because a pipeline boundary may carry an ordered tensor bundle rather than only the main hidden activation. Let $\mathcal{E}_{\text{Unrolled}}$ be the remote boundary edges for one Unrolled micro-batch. Its communication volume is

$$V_{\text{comm}}^{\text{Unrolled}} = M \sum_{e\in\mathcal{E}_{\text{Unrolled}}} \left[\text{bytes}(\Sigma_e) + \text{bytes}(\nabla\Sigma_e)\right]. \tag{28}$$

For Loop-PP, let $\mathcal{E}_{\text{within}}^{(t)}$ be the remote semantic edges within recurrent step t, and let $\mathcal{E}_{\text{feedback}}$ be the remote feedback edges between recurrent steps. Its communication volume is

$$\begin{aligned} V_{\text{comm}}^{\text{Loop-PP}} = M\Bigg[&\sum_{t=0}^{\mathcal{T}-1} \sum_{e\in\mathcal{E}_{\text{within}}^{(t)}} \left(\text{bytes}(\Sigma_e) + \text{bytes}(\nabla\Sigma_e)\right) \\ &+ \sum_{e\in\mathcal{E}_{\text{feedback}}} \left(\text{bytes}(\Sigma_e) + \text{bytes}(\nabla\Sigma_e)\right)\Bigg]. \end{aligned} \tag{29}$$

Here Σ_e may contain the hidden activation, residual source, loop state, core input, injection source, or model-specific precomputed state. Payload localization keeps fields local only when they have no remote consumer; it cannot remove fields required by a remote semantic dependency. It therefore reduces but does not eliminate the additional payload carried by recurrent execution. The repeated traversals, feedback transfers, and remotely required payload fields are imposed by the recurrent topology. They constitute a *structural communication tax*: scheduling and payload localization can reduce its exposed cost, but cannot eliminate the required communication without changing model semantics or placement.

This structural tax becomes more prominent as compute chunks become thinner. Payload size is primarily determined by micro-batch size, sequence length, hidden size, and the tensor fields in Σ_e, whereas chunk compute decreases with the number of layers assigned to that chunk. The communication intensity of an edge can therefore be characterized by

$$I_e = \frac{\mathrm{bytes}(\Sigma_e) + \mathrm{bytes}(\nabla\Sigma_e)}{\mathrm{FLOPs}(C_{\mathrm{chunk}})}. \tag{30}$$

Increasing PP or VPP generally reduces $\mathrm{FLOPs}(C_{\mathrm{chunk}})$ without proportionally reducing the payload, thereby increasing I_e. Moreover, each feedback edge lies on the recurrent critical path: the first stage of step $t+1$ cannot consume its recurrent state before the final stage of step t produces and transfers it. These properties explain why recurrent communication contributes more strongly to exposed pipeline time at fine pipeline granularity. The explicit POST/WAIT actions safely realize these dependencies; their control cost is secondary to the repeated cross-stage traversal, required payloads, and increasing communication intensity.

F LOCKSTEP BASELINE

Lockstep is the implementation's correctness-oriented schedule for the same Semantic Graph used by the main compiler. It does not use native VPP compute tables, the Cross-Stage Action DAG, or native-aware POST/WAIT linearization. Instead, it first constructs a dependency-ready bounded-1F1B compute order using the canonical topological order as the forward priority, and then projects that order into per-rank streams with unsplit communication actions.

F.1 GLOBAL COMPUTE ORDER

Let $\tau(v)$ be the position of semantic node v in the canonical topological order of $\mathcal{G}$. For every node, the scheduler maintains a forward dependency count $d_F(v) = |\mathrm{Pred}(v)|$ and a backward dependency count $d_B(v) = |\mathrm{Succ}(v)|$. A forward operation (F, v) becomes ready when $d_F(v) = 0$. A backward operation (B, v) becomes ready only after (F, v) has completed and the backward operations of all semantic successors of v have completed.

The scheduler also maintains the set $\mathcal{L}$ of live micro-batches and a configured live-microbatch cap W. A ready forward node is admissible when its micro-batch is already in $\mathcal{L}$ or when $|\mathcal{L}| < W$. Among admissible forward nodes, Lockstep selects the one with the smallest $\tau(v)$. Ready backward nodes are ordered lexicographically by

$$(m(v), -\tau(v), \mathrm{id}(v)), \tag{31}$$

so the oldest micro-batch is preferred and, within that micro-batch, the topologically latest ready node is processed first.

Bounded-1F1B construction. Algorithm 5 is a direct transcription of `build_bounded_1f1b_order` for the Lockstep policy. POPADMISSIBLE removes the minimum-τ forward node whose micro-batch is already live or can be admitted under W; it returns $\perp$ if no such node exists.

The emitted sequence

$$\mathcal{O}_{\mathrm{lock}} = \left[(\phi_0, v_0), \ldots, (\phi_{N_{\mathrm{op}}-1}, v_{N_{\mathrm{op}}-1})\right], \qquad \phi_i \in \{F, B\}, \tag{32}$$

is the Lockstep Global Compute Order. The construction corresponds to `build_bounded_1f1b_order` with `schedule_policy="lockstep"`: the policy affects the priority of dependency-ready forward nodes, while semantic readiness and the live-microbatch cap determine admissibility.

Algorithm 5 Lockstep Global Compute Order

Require: Semantic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, live-microbatch cap W

$\tau(v) \leftarrow$ position of v in TOPOLOGICALORDER($\mathcal{G}$)
$d_F(v) \leftarrow |\operatorname{Pred}(v)|$ and $d_B(v) \leftarrow |\operatorname{Succ}(v)|$ for all v
$Q_F \leftarrow \{v : d_F(v) = 0\}$ keyed by $\tau(v)$; $Q_B \leftarrow \varnothing$
$\mathcal{L} \leftarrow \varnothing$; $\mathcal{D}_F \leftarrow \varnothing$; $\mathcal{D}_B \leftarrow \varnothing$; $\mathcal{O}_{\text{lock}} \leftarrow [\,]$
$R_m \leftarrow |\{v : m(v) = m\}|$ for every micro-batch m
while $|\mathcal{D}_B| < |\mathcal{V}|$ **do**
  $useB \leftarrow (Q_B \neq \varnothing) \wedge (|\mathcal{L}| \geq W \vee Q_F = \varnothing)$
  $v \leftarrow \perp$
  if not $useB$ **then**
    $v \leftarrow$ POPADMISSIBLE($Q_F, \mathcal{L}, W$)
  end if
  if $v \neq \perp$ **then**
    append (F, v) to $\mathcal{O}_{\text{lock}}$; add v to $\mathcal{D}_F$ and $m(v)$ to $\mathcal{L}$
    for all $(v, w) \in \mathcal{E}$ **do**
      $d_F(w) \leftarrow d_F(w) - 1$
      if $d_F(w) = 0$ **then**
        insert w into Q_F keyed by $\tau(w)$
      end if
    end for
    if $d_B(v) = 0$ **then**
      insert v into Q_B keyed by $(m(v), -\tau(v), \mathrm{id}(v))$
    end if
  else
    if $Q_B = \varnothing$ **then**
      error: no dependency-ready operation
    end if
    $v \leftarrow$ POPMIN(Q_B)
    append (B, v) to $\mathcal{O}_{\text{lock}}$; add v to $\mathcal{D}_B$
    $R_{m(v)} \leftarrow R_{m(v)} - 1$
    if $R_{m(v)} = 0$ **then**
      remove $m(v)$ from $\mathcal{L}$
    end if
    for all $(u, v) \in \mathcal{E}$ **do**
      $d_B(u) \leftarrow d_B(u) - 1$
      if $d_B(u) = 0 \wedge u \in \mathcal{D}_F$ **then**
        insert u into Q_B keyed by $(m(u), -\tau(u), \mathrm{id}(u))$
      end if
    end for
  end if
end while

return $\mathcal{O}_{\text{lock}}$

F.2 COMMUNICATION PLANS

The implementation extracts the forward and backward node subsequences from $\mathcal{O}_{\text{lock}}$ and constructs one communication plan for each direction, using the same five-tuple as Equation 8. For a semantic edge $e = (u, v, \Sigma_e)$ whose endpoints reside on different PP stages,

$$\gamma_e^F = \big(e, F, p(u), p(v), \Sigma_e\big), \qquad \gamma_e^B = \big(e, B, p(v), p(u), \nabla\Sigma_e\big). \tag{33}$$

The producer x and consumer y of a transfer are (u, v) when $\phi = F$ and (v, u) when $\phi = B$, matching Equation 9. Edges whose endpoints share a PP stage remain local dependencies and do not create transfers.

Remote edges are grouped into receive-triggered batches according to the consumer y. Batches follow the corresponding forward or backward node order, while transfers in a batch follow semantic-

Algorithm 6 Lockstep Per-Rank Action Stream

Require: $\mathcal{O}_{\text{lock}}$, forward/backward plans Π_F, Π_B, stage s
$\mathcal{Q}_s \leftarrow [\,]$; $\mathcal{M}_{\text{seen}} \leftarrow \varnothing$
for all $(\phi, v) \in \mathcal{O}_{\text{lock}}$ in order **do**
  if $\phi = F \wedge m(v) \notin \mathcal{M}_{\text{seen}}$ **then**
    append EnsureBatch(v) to $\mathcal{Q}_s$
    add $m(v)$ to $\mathcal{M}_{\text{seen}}$
  end if
  $\Pi \leftarrow \Pi_F$ if $\phi = F$, otherwise Π_B
  for all $\gamma \in$ TriggeredBatch(Π, v) **do**
    if $s = \text{src}(\gamma) \vee s = \text{dst}(\gamma)$ **then**
      append Transfer(γ) to $\mathcal{Q}_s$
    end if
  end for
  if $p(v) = s$ **then**
    append Compute(ϕ, v) to $\mathcal{Q}_s$
  end if
end for

return $\mathcal{Q}_s$

edge order. As in Equation 19, the planner then assigns o_γ^{ch}, o_γ^{peer}, and κ_γ. Main-channel transfers between adjacent PP stages use the adjacent transport; feedback, injection, and non-adjacent transfers use direct peers.

F.3 Per-Rank Action Streams

For every pipeline stage s, the implementation scans $\mathcal{O}_{\text{lock}}$ once and constructs a stream $\mathcal{Q}_s$. Algorithm 6 gives the exact projection implemented by `build_phased_stage_action_stream` when transfer splitting and backward/wgrad splitting are disabled.

Equivalently, the local stream preserves the following order around every remote edge. For γ associated with $e = (u, v, \Sigma_e)$, let $(x, y) = (u, v)$ when $\phi = F$ and $(x, y) = (v, u)$ when $\phi = B$. Then

$$\text{Compute}^{\phi}(x) \prec \text{Transfer}(\gamma) \prec \text{Compute}^{\phi}(y). \tag{34}$$

The sender therefore produces a payload before its transfer action, and the receiver completes that action before executing its consumer. The implementation validates both this ordering and the per-channel and per-peer sequence numbers.

In the evaluated Lockstep configuration, communication remains an unsplit Transfer action: the baseline does not construct separate POST and WAIT vertices, does not invoke `build_stage_local_action_dag`, and does not apply native P2P pairing. Thus $\{\mathcal{Q}_s\}$ is obtained by direct validated projection of $\mathcal{O}_{\text{lock}}$, rather than by the communication-aware Action-DAG linearization used by our main method. This design makes Lockstep a simple bounded-memory reference for semantic correctness, but its canonical forward priority does not reproduce native VPP compute cadence and consequently exposes substantially less pipeline concurrency.

G Experimental Details

G.1 Model Architectures and Main-Result Settings

Model architecture. Table 2 summarizes the architectures used in the main matched-configuration comparison. Here D_{uniq} is the number of physically distinct Transformer layers stored and optimized by Loop-PP, whereas D_{eff} is the total number of logical layer visits in one forward pass after recurrence. The model names 8B-A1B, 30B-A3B, and 60B-A5B refer to the unique total and activated parameter scales and do not count a recurrently visited layer multiple times. d_{model}, d_{FFN}, and d_{expert} denote the model width, dense-FFN intermediate width, and per-expert intermediate width;

Table 2: Model architectures used in the main matched-configuration results in Figure 3.

Model	D_{uniq}	D_{eff}	d_{model}	d_{FFN}	H	G	d_{kv}	d_{expert}
8B-A1B	32	64	1280	3840	16	8	128	512
30B-A3B	48	96	2048	8192	32	8	128	768
60B-A5B	48	96	2304	9216	32	8	128	1280

H, G, and d_{kv} denote the numbers of attention heads and query groups and the configured key/value channel width.

All three models use grouped-query attention with QK layer normalization, full rotary position embeddings with base 100,000, RMSNorm, SwiGLU, zero attention and hidden dropout, and bias-free linear layers. They use vocabulary size 129,280, 128 routed experts, top-8 sigmoid routing, sequence-level auxiliary load balancing, and all-to-all token dispatch. The first Transformer layer is dense and the remaining layers are MoE layers.

Main-result training settings. Table 3 reports the training settings used specifically for the main matched-configuration results in Figure 3; later ablations state their own settings separately. We choose the global batch size (GBS), micro-batch size (MBS), and recomputation policy separately for each model size to utilize the memory and compute capacity of the 32 GPUs while keeping most compared methods feasible. For example, the 30B Tied and Unrolled baselines run out of memory without MoE recomputation. The 60B PP2 Tied and Unrolled baselines remain out of memory even under the aggressive recomputation policy used in the main experiment, indicating that their expanded parameterization does not fit this PP2 configuration.

Table 3: Training settings for the main matched-configuration results.

Model	Sequence length	GBS	MBS	Activation recomputation
8B-A1B	4096	512	2	none
30B-A3B	4096	512	2	MoE
60B-A5B	4096	256	1	MoE activations, QKV, RMSNorm

All these runs use four nodes with 32 GPUs, BF16 arithmetic, $\mathcal{T} = 2$, tensor- and context-parallel degrees of one, pipeline degrees in $\{2, 4, 8\}$ and VPP2. PP2 and PP4 use expert-parallel degree eight, while PP8 uses expert-parallel degree four. The global and micro-batch sizes are chosen so that the number M of micro-batches per data-parallel replica satisfies $M/P = 8$ at every pipeline degree.

Hardware and software environment. All main experiments use four homogeneous compute nodes, each with eight GPUs and approximately 140 GB of memory per GPU, for 32 GPUs in total. GPUs within a node communicate through a high-bandwidth all-to-all interconnect, while nodes communicate over a high-bandwidth RDMA-capable network. The software stack uses CUDA 13.2, NCCL 2.29.7, Transformer Engine 2.18.0, and Megatron Core 0.16.0; Loop-PP is implemented on top of this Megatron version. Unless otherwise stated, all compared methods use the same hardware, communication libraries, and base training stack.

G.2 Baseline Realizations

Loop-PP. The complete stack of D_{uniq} physical layers forms the recurrent region and is executed twice, denoted by $0 + D_{\text{uniq}} \times 2 + 0$ with $\mathcal{T} = 2$. Every recurrent visit references the same layer and parameter objects. The Loop-PP implementation used in our experiments enables semantic supernodes and local residual routing, constructs the native-guided phased order and split POST/WAIT Action DAG, uses native P2P pairing with a one-send/one-receive cap and static-shape metadata cache, and executes integrated whole-layer backward from immutable per-rank streams. Parameter gradients accumulate through ordinary autograd fan-in into the single physical copy; no cross-replica parameter synchronization is required.

Unrolled. The recurrent computation is materialized as independent physical layers and executed by native interleaved 1F1B. Specifically, the 8B model is expanded from 32 to 64 layers, while the

30B and 60B models are each expanded from 48 to 96 layers. This baseline matches the effective logical depth and operator count, but intentionally does not match the unique parameter count. Each logical visit owns independent parameters, gradients, and optimizer state. The experiment launcher explicitly expands the logical MoE-layer frequency so that corresponding visits execute the same dense or MoE operator type, but it does not invoke the Semantic Graph, recurrent communication planner, or Action-DAG runtime. Communication follows Megatron's native pipeline P2P path.

Tied. The Tied realization uses the same full-loop structure as Loop-PP: $0 + 32 \times 2 + 0$ for 8B and $0 + 48 \times 2 + 0$ for 30B and 60B. It first materializes the effective feed-forward depth so that Megatron's native interleaved scheduler sees an ordinary expanded layer sequence. The two temporal copies of each layer are separate physical module and parameter objects annotated by a common logical tie key; the first copy is canonical. At initialization or finetuning load, the canonical parameters and router expert-bias buffers are copied to every temporal replica, including replicas placed on other pipeline stages. After backward, gradients with the same tie key are accumulated in reverse loop order in FP32 and pipeline-reduced when their replicas span stages. The resulting logical gradient is written to each replica before the normal data/tensor-parallel reduction. Only the canonical parameter is admitted to the optimizer, so noncanonical replicas do not own Adam state; after the optimizer step, the updated canonical value is copied back to all replicas. Distributed-checkpoint metadata marks noncanonical copies as replicas and omits their duplicate parameter shards while retaining genuinely unique output adapters and other untied state. It retains expanded module, activation, gradient-buffer, and synchronization overhead, so it is not equivalent to Loop-PP's single physical parameterization.

Lockstep. Lockstep uses the same full-loop Semantic Graph as Loop-PP but applies the conservative compiler in Appendix F. It chooses dependency-ready forward work in canonical topological order, switches to reverse work under a live-microbatch bound, and projects the resulting global order with unsplit blocking transfer actions. It does not use native VPP slot tables, split POST/WAIT actions, native P2P epochs, semantic supernodes, or the communication-aware stage-local Action DAG. For Figure 3, its live-microbatch window is set to $W = PV$ at PP2 and PP4, matching the activation concurrency of native interleaved 1F1B. At PP8, $W = PV = 16$ runs out of memory, so we reduce the window to eight. This smaller window also explains why Lockstep uses less peak memory than Tied and Unrolled at PP8.

G.3 Measurement and Data Provenance

Unless otherwise specified, all reported throughput in the paper uses the same protocol. We run each method for 50 training steps and take T_{step} as the mean over the last 20 steps; each step uses the maximum elapsed time over all ranks. Throughput is $\text{GBS} \times S/T_{\text{step}}$ for sequence length S (the evaluated workloads use $S = 4096$). Peak allocated memory is the maximum over all ranks. For failed 60B PP2 baselines, Figure 3 reports only an OOM marker and does not treat the last pre-failure allocator sample as a successful-run peak. The main figure reports the product-table runs with random initialization and zero learning rate. This controlled setting isolates execution-system behavior from checkpoint- and convergence-dependent effects. Each measured step still executes the complete forward, backward, gradient-finalization, and optimizer paths. These runs evaluate systems throughput rather than model quality; semantic behavior under nonzero-learning-rate training is evaluated separately in Appendix H.

G.4 Timing Decomposition and Instrumentation

Instrumentation boundary. The phase profiler is enabled only for selected steady-state iterations. It records CUDA events around the training-step boundary, the pipeline-schedule invocation, local forward/backward action execution, pipeline receive/wait paths, gradient finalization, and the optimizer step. Events are synchronized once at the end of the profiled iteration; no synchronization is inserted between individual actions. Consequently, the reported quantities are elapsed CUDA intervals on the measured rank rather than sums of CPU wall-clock timers.

End-to-end and pipeline time. T_{step} spans the profiled training iteration from the step-start event to the step-end event. For end-to-end throughput, each iteration takes the maximum T_{step} over all

ranks, and throughput is GBS $\times S/T_{\text{step}}$ for sequence length S. T_{pipe} spans entry to and exit from the pipeline schedule. For a detailed phase row, the analysis first selects the rank with the largest T_{pipe} in that iteration and then reads every component from that same rank and iteration. It never combines the step time from one rank with phase components from another.

Local actions. T_{local} is the sum of CUDA intervals enclosed by locally executed forward, backward, and, when present, separate weight-gradient actions:

$$T_{\text{local}} = T_F + T_B + T_W.$$

T_F is the service time of forward actions, T_B is the service time of backward actions, and T_W is the service time of separately scheduled weight-gradient actions. These scopes include all work performed inside the action, such as attention and FFN kernels, router and expert kernels, MoE dispatch/combine collectives, activation recomputation triggered by backward, autograd execution, and launch gaps internal to the action. In particular, expert-parallel communication executed inside an MoE forward or backward action is included in T_F or T_B; T_{local} is therefore not pure GPU computation. Thus, T_{local} is *local-action service time*, not pure arithmetic or kernel-active time. In our implementation, weight-gradient computation is integrated into ordinary backward, so $T_W = 0$ does not mean that weight gradients are absent; their cost is included in T_B.

Exposed pipeline time. The schedule-independent first-order measure of pipeline cost exposed on the critical path is

$$T_{\text{exposed}} = T_{\text{pipe}} - T_{\text{local}}.$$

It measures all pipeline time not covered by local forward, backward, or weight-gradient service. This interval includes exposed destination receives and waits, dependency stalls, communication posting and readiness checks, sender-side waits, action dispatch, launch gaps between actions, tensor-state bookkeeping, uninstrumented stream synchronization, and other runtime gaps on the pipeline critical path. It does not equal total communication time because communication nested inside a local action or overlapped with local work is already covered by that action. Only part of T_{exposed} is a literal idle bubble. When comparing Loop-PP with Unrolled, we therefore use $\Delta T_{\text{exposed}} = T_{\text{exposed}}^{\text{Loop-PP}} - T_{\text{exposed}}^{\text{Unrolled}}$ This difference is a useful measure of the additional pipeline scheduling tax paid by Loop-PP, but it combines dependency stalls, exposed communication, and runtime gaps rather than isolating scheduler overhead alone.

Pipeline-external tail. The remaining interval is

$$T_{\text{tail}} = T_{\text{step}} - T_{\text{pipe}} = T_{\text{finalize}} + T_{\text{optimizer}} + T_{\text{other}}.$$

T_{finalize} contains gradient finalization and associated distributed reductions; $T_{\text{optimizer}}$ is the optimizer-step interval; and T_{other} is the nonnegative residual at the step boundary, including work not enclosed by the other scopes and synchronization that crosses the pipeline-profiler boundary. Unlike T_W, which measures a separately scheduled layer-local weight-gradient action inside the pipeline schedule, T_{finalize} covers post-pipeline gradient finalization and distributed reductions. The tail is a measured time component, not a tunable scheduling parameter. It can differ substantially between Loop-PP and Unrolled because the latter owns more unique parameters, optimizer state, and gradient-finalization work.

Additivity and nested observations. Every reported phase row is constructed to satisfy the additive relations

$$T_{\text{step}} = T_{\text{pipe}} + T_{\text{tail}}, \qquad T_{\text{pipe}} = T_{\text{local}} + T_{\text{exposed}},$$

up to event-resolution and printed-rounding error. MoE all-to-all scopes are nested observations inside T_F or T_B. They are useful for diagnosing local actions but must not be added again to T_{local}, and they cannot be called exposed expert-parallel wait without a separate activity-level interval-union analysis.

G.5 PP8-Cliff Mechanism Details

The phase decomposition in Figure 4(a–b) comes from the repaired-schedule 30B paired profile on two nodes and 16 GPUs. Both PP2 and PP8 use sequence length 4,096, GBS 256, MBS 2,

Table 4: Complete data plotted in Figure 3. Each entry is throughput in tokens/s followed by peak allocated memory in GiB.

Model	PP	Loop-PP	Unrolled	Tied	Lockstep
8B	2	325,903 / 52.3	300,787 / 62.9	287,647 / 63.6	171,844 / 96.2
8B	4	276,061 / 50.4	266,644 / 62.1	239,370 / 63.7	77,942 / 109.0
8B	8	237,796 / 58.4	258,406 / 62.7	250,369 / 64.6	37,902 / 74.7
30B	2	131,387 / 69.3	116,931 / 97.1	109,074 / 97.7	70,230 / 106.6
30B	4	110,236 / 67.1	101,642 / 89.2	105,551 / 89.8	34,590 / 116.8
30B	8	103,465 / 79.8	110,183 / 91.3	109,021 / 92.1	16,881 / 85.6
60B	2	96,326 / 83.8	OOM	OOM	OOM
60B	4	83,860 / 73.8	83,360 / 109.8	81,013 / 111.4	24,448 / 116.1
60B	8	72,248 / 80.5	81,856 / 110.4	75,804 / 110.9	11,493 / 82.5

MoE recomputation, and learned routing. PP2 uses expert parallelism eight and PP8 uses expert parallelism two. For each profiled iteration, all phase components are taken from the rank with the largest pipeline time; Table 5 reports a representative additive sample from the same rank and iteration.

Table 5: Phase decomposition used in Figure 4(a–b). All times are seconds.

PP	Method	T_{step}	T_{local}	T_{exposed}	T_{tail}
2	Loop-PP	17.108	16.494	0.553	0.061
2	Unrolled	17.366	16.744	0.170	0.452
8	Loop-PP	21.160	16.547	4.565	0.048
8	Unrolled	17.836	16.849	0.289	0.698

Figure 4(c) uses the product-tuned compute-granularity sweep. Within each PP configuration at virtual pipeline parallelism two, Loop-PP is paired with Unrolled under the same GBS, MBS, model shape, and expert-parallel degree. The throughput points are listed in Table 6.

Table 6: VPP2 throughput in Figure 4(c), in tokens/s.

PP	Loop-PP	Unrolled
2	133,443	125,465
4	108,069	115,696
8	64,721	99,694

To further diagnose the source of exposed pipeline time, we instrument each Loop-PP destination WAIT. For this diagnostic only, $T_{\text{PP-wait}}$ denotes the CUDA time inside explicitly instrumented destination-side receive/WAIT actions. It is an internal diagnostic component of T_{exposed}, not total communication time or the complete exposed pipeline interval. Each WAIT is classified from the rank-local ready set at the moment it begins. A WAIT is *dependency-limited* when no later local compute action is semantically ready, *linearization-limited* when such an action exists but the compiled local order prevents using it, and *residual* when neither classification accounts for the measured interval. For every selected record,

$$T_{\text{PP-wait}} = T_{\text{dep}} + T_{\text{lin}} + T_{\text{residual}}.$$

For each iteration, the analysis selects the rank with maximum T_{pipe} and reads all four quantities from that same rank. Table 7 reports time-weighted shares on that rank.

Taken together, the three diagnostics identify exposed recurrent dependencies as the main source of the PP8 slowdown. Relative to Unrolled, Loop-PP saves $0.302\,\text{s}$ of local-action time and $0.650\,\text{s}$ of tail time at PP8, but adds $4.276\,\text{s}$ of exposed pipeline time, increasing total step time by $3.324\,\text{s}$. Increasing the Loop-PP chunk thickness from three layers at VPP2 to six layers at VPP1 raises throughput from 64.7k to 71.3k tokens/s. Thus, a thicker compute window amortizes part of the waiting cost but does not remove the recurrent critical path. Meanwhile, 99.98% of the explicitly instrumented PP8 destination-WAIT time is dependency-limited and none is linearization-limited, indicating that the dominant waits arise from semantic readiness rather than suboptimal static action ordering. This classification applies only to the explicitly instrumented destination waits and does not attribute the remainder of T_{exposed} to the same cause.

Table 7: Explicit destination-wait classification on the Loop-PP pipeline-critical rank in the product-tuned compute-granularity sweep.

Topology	$T_{\text{PP-wait}}$ (s)	Dependency	Linearization	Residual
PP2	0.086	99.40%	0.00%	0.60%
PP4	1.284	99.92%	0.00%	0.08%
PP8	4.176	99.98%	0.00%	0.02%

G.6 Recurrent-Structure Sweep Details

Figure 6 uses the 8B workload on four nodes with 32 GPUs, PP2 or PP4, VPP2, EP8, TP1, CP1, sequence length 4,096, GBS 512, MBS 2, random initialization, and zero learning rate. It retains 32 unique layers, while Unrolled materializes one independent physical layer for every logical visit. The full-loop $\mathcal{T} = 2$ points reuse the canonical 8B PP2 and PP4 measurements from Figure 3. For the fixed-effective-depth comparison, the same canonical 64-layer Unrolled measurements serve as the common denominator for all three recurrent layouts. Tables 8 and 9 report the corresponding measurements.

Figure 6(a) uses the full-loop layout $0 + 32 \times \mathcal{T} + 0$ and varies $\mathcal{T}$ from two to four at PP2 and PP4. The $\mathcal{T} \leq 3$ runs use no activation recomputation. At $\mathcal{T} = 4$, the 128-layer Unrolled model exceeds memory without recomputation, so both methods use the same selective recomputation of MoE activations, QKV, and layer normalization. Figure 6(b) fixes $\mathcal{T} = 2$ and varies the recurrent fraction at PP2 and PP4 using $12 + 8 \times 2 + 12$, $8 + 16 \times 2 + 8$, $4 + 24 \times 2 + 4$, and $0 + 32 \times 2 + 0$. These layouts contain the same 32 unique layers but have effective depths of 40, 48, 56, and 64, respectively. Figure 6(c) instead fixes the effective depth at 64 and compares $0 + 32 \times 2 + 0$, $8 + 16 \times 3 + 8$, and $12 + 8 \times 5 + 12$ at both PP2 and PP4. For the last layout, the reported point uses the balanced PP-stage placement that avoids concentrating all repeated core layers on one physical rank.

Table 8: Full-loop depth sweep used in Figure 6(a).

PP	$\mathcal{T}$	Loop-PP TPS	Unrolled TPS	Relative throughput
2	2	325,903	300,787	1.084
2	3	226,588	202,378	1.120
2	4	148,483	143,586	1.034
4	2	276,061	266,644	1.035
4	3	197,531	192,899	1.024
4	4	127,097	140,271	0.906

Table 9: Recurrent-fraction and fixed-effective-depth points used in Figure 6(b–c). Relative throughput is Loop-PP/Unrolled.

Panel	Configuration	PP2	PP4	Effective depth
(b)	25% recurrent	0.734	0.654	40
(b)	50% recurrent	1.054	0.994	48
(b)	75% recurrent	0.931	0.670	56
(b)	100% recurrent	1.084	1.035	64
(c)	$0 + 32 \times 2 + 0$	1.084	1.035	64
(c)	$8 + 16 \times 3 + 8$	1.055	1.036	64
(c)	$12 + 8 \times 5 + 12$	0.947	0.929	64

Why recurrent depth is not monotonic. Across the three panels, relative performance is determined not only by effective computation, but also by the number of recurrent traversals, the placement of the recurrent region, and the amount of independent computation available around each cross-stage recurrent dependency. In Figure 6(a), Loop-PP retains 32 unique layers as $\mathcal{T}$ grows, whereas Unrolled expands from 64 to 128 physical layers. More recurrence can therefore strengthen Loop-PP's potential savings in parameter, gradient-finalization, and optimizer-related work; this is consistent with the modest increase from $\mathcal{T} = 2$ to 3 at PP2. At the same time, every additional traversal lengthens each microbatch's feedback-dependency chain and revisits the relevant

PP boundaries. These dependencies become harder to hide when PP increases and local compute chunks become thinner. At $\mathcal{T} = 4$, the structural cost remains largely amortized at PP2, but dominates at PP4, where relative throughput falls to 0.906. This point also includes the effect of selective recomputation: the 128-layer Unrolled model exceeds memory without it, so both methods use the same MoE-activation, QKV, and layer-normalization recomputation policy.

Why recurrent fraction is placement-sensitive. Figure 6(b) is non-monotonic because changing recurrent fraction moves the boundaries among the prelude, recurrent region, and coda relative to the cyclic VPP chunks. Chunks covered by the recurrent region execute twice, whereas prelude and coda chunks execute once. The boundary locations therefore change the per-stage effective work, the PP boundaries crossed by feedback edges, and the lifetimes of communicated payloads. Under the PP4/VPP2 cyclic placement used here, the 50% and 100% layouts distribute repeated work more evenly across PP stages, while the 25% and 75% layouts create a longer stage-critical path. Measured in equivalent layer visits, the per-stage work for recurrent fractions 25%, 50%, 75%, and 100% is respectively $[12, 8, 8, 12]$, $[12, 12, 12, 12]$, $[12, 16, 16, 12]$, and $[16, 16, 16, 16]$. Their recurrent boundaries also expose feedback at different PP/VPP boundaries. Consequently, 50% and 100% outperform their neighboring fractions. Recurrent fraction alone is therefore insufficient to predict performance; how the recurrent region maps to pipeline chunks is equally important. As in Appendix B, the native interleaved table requires an equal logical-chunk count on every PP stage. Under the PP4/VPP2 cyclic placement used here, this holds for the 50% and 100% layouts but not for 25% and 75%, which therefore use the dependency-ready bounded-1F1B compute template rather than the native table. Under PP2/VPP2, all four layouts keep equal chunk counts and remain on the native table. PP2 therefore provides the controlled evidence for the non-monotonic trend, while PP4 reflects the combined effects of placement and the template switch.

Why traversal count matters at fixed effective depth. Figure 6(c) holds effective depth at 64 and uses the same canonical 64-layer Unrolled measurements for normalization. The three Loop-PP layouts contain two, three, and five recurrent traversals, respectively. Moving from $0 + 32 \times 2 + 0$ to $8 + 16 \times 3 + 8$ shortens each recurrent region but increases the number of step-to-step feedback transitions from one to two. The additional dependency slightly lowers relative throughput at PP2, while the two layouts are nearly tied at PP4, where placement of the shorter core approximately offsets the extra traversal. In contrast, $12 + 8 \times 5 + 12$ performs only eight recurrent layers per traversal but requires four feedback transitions. Even with the balanced placement used for the reported point, it has more frequent feedback, thinner compute fragments, and more cross-stage payload lifetimes. Less independent computation is available around each transfer, so dependency stalls are more likely to remain exposed; relative throughput consequently falls to 0.947 at PP2 and 0.929 at PP4. Thus, at matched effective computation, fewer traversals and thicker recurrent chunks are generally preferable. The structural cost is governed jointly by traversal frequency, placement-sensitive cross-stage feedback, and the computation available to overlap it.

G.7 Equal-48-Layer No-Recomputation Study

This auxiliary study evaluates the 30B model geometry at a shallower but compute-matched effective depth. Loop-PP and Lockstep execute 24 unique layers for two recurrent steps, Unrolled materializes 48 independent layers, and Tied materializes a weight-tied 48-layer expansion. All runs use four nodes with 32 GPUs, sequence length 4,096, GBS 512, MBS 2, expert parallelism four, random initialization, zero learning rate, and no activation recomputation. The PP2 Lockstep point uses the passing two-microbatch live window after its larger-window run exceeded memory capacity. Table 10 reports throughput and peak memory.

Table 10: Equal-48-layer no-recomputation results. Each entry reports throughput in tokens/s followed by peak allocated memory in GiB.

PP	Loop-PP	Unrolled	Tied	Lockstep
2	333,016 / 68.3	318,323 / 90.1	300,917 / 91.9	177,502 / 73.7
4	277,206 / 64.4	294,427 / 82.2	264,334 / 86.0	84,064 / 68.9
8	237,793 / 72.0	252,225 / 77.4	235,680 / 81.9	42,428 / 75.9

The result provides two pieces of supporting evidence. First, parameter reuse reduces peak allocated memory relative to Unrolled by 24.2%, 21.7%, and 7.0% at PP2, PP4, and PP8, respectively, even without activation recomputation. Second, the PP-dependent throughput trend remains visible at the shallower effective depth: Loop-PP is 4.6% faster than Unrolled at PP2 but 5.8% and 5.7% slower at PP4 and PP8. Loop-PP also outperforms Tied at all three pipeline degrees and remains substantially faster than Lockstep. This experiment supports the robustness of the memory-saving and pipeline-granularity observations.

Comparison with the 96-layer product workload. Table 11 compares only dimensionless quantities derived from the equal-48-layer study and the 30B rows of the product table in Table 4. The product values are taken directly from that table rather than from an older 96-layer analysis. Because the two workloads differ in effective depth and recomputation regime, we do not compare their absolute throughput.

Table 11: Cross-depth comparison. $\eta_{\text{48-layer}}$ and $\eta_{\text{96-layer}}$ are Loop-PP/Unrolled throughput in the equal-48-layer study and the 30B product (96-layer effective) workload, respectively; memory saving is the relative reduction in peak allocated memory.

PP	$\eta_{\text{48-layer}}$	Mem. save	$\eta_{\text{96-layer}}$	Mem. save
2	1.046	24.2%	1.124	28.6%
4	0.942	21.7%	1.085	24.8%
8	0.943	7.0%	0.939	12.6%

Both depths show the same two qualitative properties. First, Loop-PP reduces peak memory at every pipeline degree, including by 24.2% at PP2 in the shallower no-recomputation workload. Thus, the capacity benefit is not restricted to an extremely deep effective model or created only by activation recomputation. Second, Loop-PP's relative throughput declines as the pipeline is divided more finely: it leads at PP2 and loses its advantage at larger PP in the 48-layer study, while the 96-layer product workload changes from a PP2/PP4 advantage to a PP8 deficit. This shared trend supports pipeline granularity, rather than effective depth alone, as the relevant performance factor.

G.8 Dense/MoE Details

Dense and MoE comparison. This experiment asks whether Loop-PP's throughput advantage is specific to MoE or also appears on a dense Transformer with the same backbone. We therefore keep the 30B attention width, layer count, and recurrent layout, and replace only the MoE block with a dense SwiGLU FFN of intermediate width 6,144 ($3d_{\text{model}}$), a standard dense-FFN ratio for this hidden size. Expert parallelism is disabled on the dense workload so that the comparison isolates operator type rather than expert-parallel placement. Dense runs use no recomputation because they fit in memory; MoE runs recompute the MoE module, which is required for the same sequence length and batch. Figure 5 uses four nodes with 32 GPUs, PP2 or PP4, VPP2, TP1, CP1, sequence length 4,096, GBS 512, MBS 2, and a 30B-shape model with 48 unique layers executed for two recurrent steps. The model uses $d_{\text{model}} = 2048$, 32 attention heads, eight query groups, and $d_{\text{kv}} = 128$. The MoE workload uses 128 experts, top-8 routing, expert width 768, and expert parallelism four. The paired runs use distributed optimization without gradient-reduction or parameter-gather overlap and disable CUDA graphs. Table 12 reports the plotted throughput and phase times.

Table 12: Dense/MoE throughput and phase decomposition used in Figure 5. Times are seconds; tail is $T_{\text{step}} - T_{\text{pipe}}$. A2A scope is a nested observation inside T_{local} and is not an additive component.

Workload	Method	Throughput	T_{step}	T_{local}	T_{exposed}	T_{tail}	A2A scope
Dense PP2	Loop-PP	340,362	6.162	5.349	0.772	0.041	0.000
Dense PP2	Unrolled	342,080	6.131	5.542	0.518	0.071	0.000
MoE PP2	Loop-PP	134,827	15.554	14.192	1.071	0.291	3.451
MoE PP2	Unrolled	125,817	16.668	15.547	0.564	0.557	4.195
Dense PP4	Loop-PP	292,529	7.169	5.163	1.981	0.025	0.000
Dense PP4	Unrolled	310,386	6.757	5.299	1.417	0.041	0.000
MoE PP4	Loop-PP	114,187	18.366	14.683	3.620	0.063	3.779
MoE PP4	Unrolled	113,121	18.539	15.137	3.289	0.113	4.498

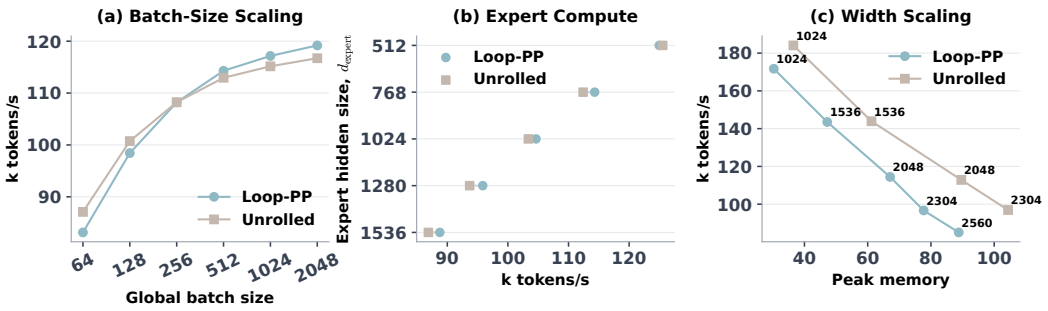

Figure 7: Sensitivity of the 30B MoE workload. (a) Throughput versus global batch size. (b) Throughput versus expert hidden size. (c) Throughput–memory trade-off as model width increases; Unrolled runs out of memory at $d_{\text{model}} = 2560$.

G.9 Workload and Compute-Intensity Sensitivity

This section varies global batch size, expert width, and model width on the 30B MoE model to test whether Loop-PP's throughput gap relative to Unrolled is a fixed scheduling tax or disappears monotonically with compute intensity. We use four nodes with 32 GPUs, PP4, VPP2, EP8, TP1, CP1, sequence length 4,096, MBS 2, random initialization, zero learning rate, and MoE recomputation. The base model has 48 unique layers, $d_{\text{model}} = 2048$, dense FFN width 8,192, 128 top-8 experts, and $d_{\text{expert}} = 768$. Each sweep changes only one axis. The GBS sweep varies global batch size from 64 to 2,048 while retaining the base model. The expert-width sweep fixes GBS 512 and $d_{\text{model}} = 2048$ while varying d_{expert} from 512 to 1,536. The model-width sweep fixes GBS 512, sets $d_{\text{expert}} = 0.375 d_{\text{model}}$, dense FFN width to $4d_{\text{model}}$, the number of attention heads to $d_{\text{model}}/64$, and uses a 4:1 GQA ratio. Figure 7 shows the three sweeps; the corresponding numbers are in Tables 13–15.

Table 13: Global-batch-size sweep used in Figure 7(a).

GBS	Loop-PP TPS	Loop mem.	Unrolled TPS	Unrolled mem.
64	83,141	58.5	87,096	74.2
128	98,428	67.0	100,736	89.4
256	108,202	67.1	108,230	89.5
512	114,297	67.1	112,947	89.6
1024	117,156	67.1	115,154	89.5
2048	119,186	67.1	116,720	89.7

We first vary global batch size. As shown in Figure 7(a), increasing GBS from 64 to 2,048 raises Loop-PP throughput from 83.1k to 119.2k tokens/s. Relative to Unrolled, Loop-PP is 4.5% and 2.3% slower at GBS 64 and 128, reaches parity near GBS 256, and leads by approximately 1–2% at larger batches. Thus, a larger pipeline workload more effectively amortizes Loop-PP's fixed scheduling cost; the memory advantage remains throughout the GBS range.

Table 14: Expert-width sweep used in Figure 7(b).

d_{expert}	Loop-PP TPS	Loop mem.	Unrolled TPS	Unrolled mem.
512	124,969	62.3	125,536	80.2
768	114,314	67.1	112,403	89.6
1024	104,638	71.8	103,371	99.1
1280	95,848	76.6	93,687	108.7
1536	88,766	81.4	86,875	118.4

We next fix GBS 512 and increase d_{expert} from 512 to 1,536. Figure 7(b) shows no monotonic improvement in Loop-PP's relative throughput: the paired methods remain within approximately 0.5–2.3% throughout the sweep. Loop-PP uses less peak memory at every expert width.

Table 15: Hidden-width sweep used in Figure 7(c). An OOM row reports the last complete pre-failure allocator marker, not the failure peak.

d_{model}	Loop-PP TPS	Loop mem.	Unrolled TPS	Unrolled mem.
1024	171,718	30.3	184,081	36.5
1536	143,519	47.2	143,928	61.2
2048	114,429	67.1	112,857	89.6
2304	96,673	77.7	96,923	104.4
2560	85,051	88.8	OOM	100.5

Finally, we fix GBS 512 and scale model width proportionally. Figure 7(c) likewise shows no monotonic relative throughput trend. Loop-PP trails at $d_{\text{model}} = 1024$, is near parity at 1,536 and 2,304, and leads slightly at 2,048. Its memory usage remains lower throughout, and it reaches 85.1k tokens/s at $d_{\text{model}} = 2560$, where Unrolled runs out of memory. Together, these results show that increasing global batch size amortizes Loop-PP's fixed scheduling tax, whereas increasing per-chunk FLOPs alone does not consistently hide this structural cost. Parameter reuse provides a robust memory advantage across compute intensities and expands the feasible model-capacity envelope.

G.10 Best-Measured Configurations

Protocol and search dimensions. The equal-resource experiment fixes four nodes with 32 GPUs, sequence length 4,096, GBS 512, tensor- and context-parallel degrees one, random initialization, zero learning rate, and no checkpoint loading or saving. Loop-PP and Unrolled independently search their feasible PP/VPP, MBS, and recomputation choices, and the comparison reports the best throughput found in each search. The recorded search covers PP2 and PP4, VPP degrees two and four, MBS values one, two, and four, and architecture-valid subsets of no recomputation, MLP/MoE recomputation, full recomputation, and selective QKV, layer-normalization, attention, or MoE-activation recomputation. Invalid configurations, incomplete runs, and OOM points are excluded from the feasible candidate set.

Dense 14B search. The Dense model contains 48 unique layers executed twice by Loop-PP and 96 independent layers for Unrolled, with $d_{\text{model}} = 5120$, SwiGLU FFN width 13,824, 40 attention heads, eight query groups, vocabulary size 152,064, and untied input/output embeddings. Loop-PP was evaluated with PP=2/4, VPP=2/4, MBS=1/2/4, and no, MLP, core-attention, or full recomputation where supported. Its fastest stable point was PP=2/VPP2, MBS=1, without recomputation at 96.6k tokens/s. The PP2/VPP4 point reached 75.8k tokens/s; larger-MBS candidates commonly exceeded memory capacity. For Unrolled, PP2/VPP2 with MLP and full recomputation reached 81.9k and 76.0k tokens/s, respectively. PP4/VPP2 without recomputation ran out of memory, while PP4/VPP4 with QKV, layer-normalization, and core-attention recomputation reached the best Unrolled result of 87.0k tokens/s.

MoE 30B search. The MoE model contains 48 unique layers executed twice or 96 independent layers, with $d_{\text{model}} = 2048$, dense FFN width 8,192, 128 top-8 experts, expert width 768, and expert parallelism four. The initial twelve-point search varied PP2/PP4, MBS2/4, and no versus MoE recomputation. No-recomputation configurations failed from insufficient memory; the viable main path used MoE recomputation, reaching 135.7k tokens/s for Loop-PP and 126.4k for Unrolled at PP2. The follow-up search added VPP4 and selective recomputation. For Loop-PP at PP2/VPP4 and MBS2, QKV-only, QKV plus MoE-activation, QKV plus layer-normalization plus MoE-activation, and layer-normalization plus MoE-activation recomputation reached 165.9k, 162.9k, 159.5k, and 170.7k tokens/s, respectively; no recomputation, layer-normalization only, and MoE-activation only ran out of memory. The best Unrolled point used PP4/VPP4, MBS2, and QKV, layer-normalization, and MoE-activation recomputation, reaching 151.5k tokens/s. Because the winning methods select different recomputation policies, this is an equal-resource comparison, not an iso-recomputation comparison. Table 16 lists representative search candidates, and Table 17 records the best measured points, matching Table 1 in the main text.

Table 16: Representative candidates from the equal-resource search. Memory is peak allocated GiB; "–" denotes an unsuccessful configuration.

Model	Method	PP/VPP	MBS	Recompute	Status	Throughput	Memory
Dense 14B	Loop-PP	2/2	1	none	PASS	96.6k	113.7
Dense 14B	Loop-PP	4/2	1	none	PASS	84.6k	100.2
Dense 14B	Loop-PP	4/2	2	none	OOM	–	–
Dense 14B	Unrolled	2/2	1	MLP	PASS	81.9k	119.7
Dense 14B	Unrolled	2/2	1	full	PASS	76.0k	97.0
Dense 14B	Unrolled	4/2	1	none	OOM	–	–
Dense 14B	Unrolled	4/4	1	QKV+LN+attention	PASS	87.0k	115.7
MoE 30B	Loop-PP	2/2	2	MoE	PASS	135.7k	79.1
MoE 30B	Loop-PP	4/2	2	MoE	PASS	116.1k	71.8
MoE 30B	Loop-PP	4/2	4	MoE	PASS	117.4k	120.6
MoE 30B	Loop-PP	2/4	2	QKV	PASS	165.9k	112.0
MoE 30B	Loop-PP	2/4	2	QKV+MoE act.	PASS	162.9k	103.5
MoE 30B	Loop-PP	2/4	2	LN+MoE act.	PASS	170.7k	117.7
MoE 30B	Unrolled	2/2	2	MoE	PASS	126.4k	116.4
MoE 30B	Unrolled	2/4	2	MoE	PASS	127.9k	112.5
MoE 30B	Unrolled	4/4	2	QKV+LN+MoE act.	PASS	151.5k	112.7

Table 17: Best measured configurations under the equal 32-GPU resource budget.

Model	Method	Throughput	Memory	Speedup
Dense 14B	Loop-PP	96.6k	113.7	1.110×
Dense 14B	Unrolled	87.0k	115.7	–
MoE 30B	Loop-PP	170.7k	117.7	1.127×
MoE 30B	Unrolled	151.5k	112.7	–

H Correctness Validation

Loop-PP changes the execution plan rather than the model architecture or optimization objective. Accordingly, the evaluation below assesses semantic equivalence to recurrent execution through losses, intermediate tensors, gradients, parameter updates, and training trajectories.

H.1 Short-Horizon Numerical Comparison

We first compare Loop-PP PP2 against a PP1 recurrent oracle on an 8B MoE model with $\mathcal{T} = 2$. Both runs load the same weights, use sequence length 4,096, MBS=1, GBS=128, and set the learning rate to 10^{-4}. Table 18 reports language-model loss and auxiliary routing loss.

Table 18: Five-step PP1 and PP2 comparison for the 8B recurrent model.

Step	PP1 LM	PP2 LM	$\lvert\Delta\text{LM}\rvert$	PP1 aux	PP2 aux
1	2.285221	2.285221	0	1.594339	1.594339
2	2.341657	2.341657	0	1.588836	1.588836
3	2.297385	2.297385	0	1.630002	1.630002
4	2.237283	2.237283	0	1.588754	1.588754
5	2.329112	2.329112	0	1.577844	1.577844

At the logged precision, both losses are bit-exact for all five steps. Loop-PP at PP2 therefore reproduces the PP1 forward and loss computation on this 8B workload.

H.2 Deterministic Oracle

The 8B comparison above uses training-log scalars and does not test tensors. We therefore run a separate *deterministic oracle*: a tiny dense looped Transformer with a fixed seed, identical initial weights, and identical micro-batches. A PP1 recurrent execution first materializes the reference tensors. Loop-PP then runs the same inputs at PP2. The test compares, under explicit tolerances, per-microbatch language-model loss, every Semantic-Graph event output, every event input gradient, every mapped rank-local parameter gradient, and the weights after one SGD step. Table 19

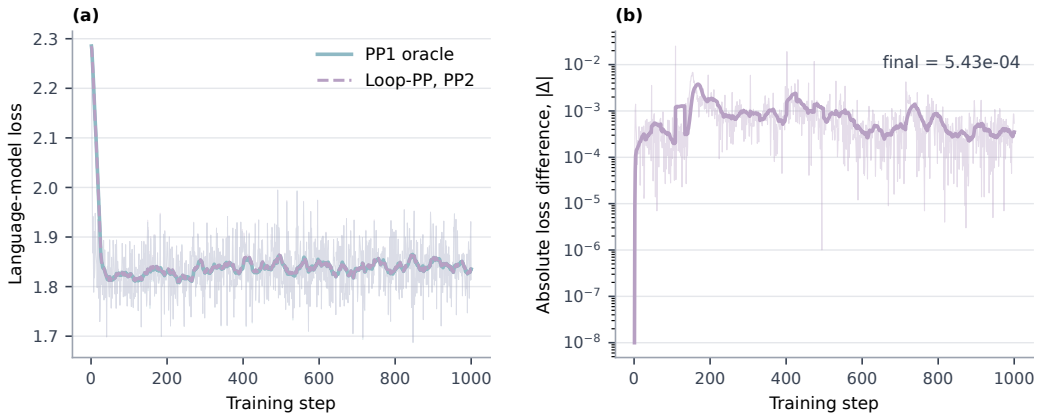

Figure 8: A 1,000-step training comparison between the PP1 recurrent oracle and Loop-PP at PP2 on the 8B recurrent model with $\mathcal{T} = 3$. Thin lines show per-step loss and thick lines show a 25-step moving average. The right panel reports the absolute per-step loss difference.

summarizes the checks. We do not use the printed global gradient norm as a criterion, because that scalar depends on distributed reduction order. As shown in Table 19, all five checks pass. Loop-PP therefore matches the PP1 oracle not only in loss, but also in event tensors, parameter gradients, and the resulting SGD update.

Table 19: Deterministic PP1 oracle versus Loop-PP at PP2.

Quantity	Tolerance	Result
Per-microbatch LM loss	$10^{-5}/10^{-6}$	PASS
Event outputs	$10^{-5}/10^{-6}$	PASS
Event input gradients	$10^{-4}/5\times10^{-5}$	PASS
Rank-local parameter gradients	$10^{-4}/5\times10^{-5}$	PASS
One-step SGD parameter update	$10^{-4}/5\times10^{-5}$	PASS

H.3 Long-Horizon Training

This run uses PP1 on four GPUs and Loop-PP at PP2 on eight GPUs, with sequence length 4,096, MBS 1, GBS 128, constant learning rate 10^{-4}, and no activation recomputation. As shown in Figure 8, the two loss trajectories remain visually superimposed. They are exactly equal at step 1 and differ by 5.43×10^{-4} at step 1,000. The corresponding endpoint losses are 1.813975 and 1.814518; the endpoint routing-load CVs are 0.677 and 0.676. Over a long training horizon, Loop-PP therefore stays numerically close to the PP1 recurrent execution and does not accumulate a growing semantic drift.