ASYRA: Automating Graph Scheduling for Communication–Computation Overlap in Efficient Model Parallelism

Lei Zhang [*1] **Zhisheng Ye** [*1]

Abstract

Scaling large models requires complex multi-dimensional (n-D) parallelism, yet this paradigm suffers from severe communication bubbles that diminish efficiency. However, existing overlapping techniques rely on complex manual kernel fusion and lack generality across diverse parallelism patterns, model architectures, or workloads. Hence, we propose ASYRA, an automatic communication–computation overlapping approach for n-D model parallelism via graph scheduling during the compiling stage. By estimating runtime makespan and memory usage via the simulator under various configurations, ASYRA applies graph-based scheduling with tiling, reordering, and bucketing for graph operators to maximize overlapping, thus achieving high efficiency. Extensive n-D parallel experiments demonstrate that ASYRA achieves up to 4% to 30% speedup in training and inference, and saves nearly 30% of the activation memory in inference.

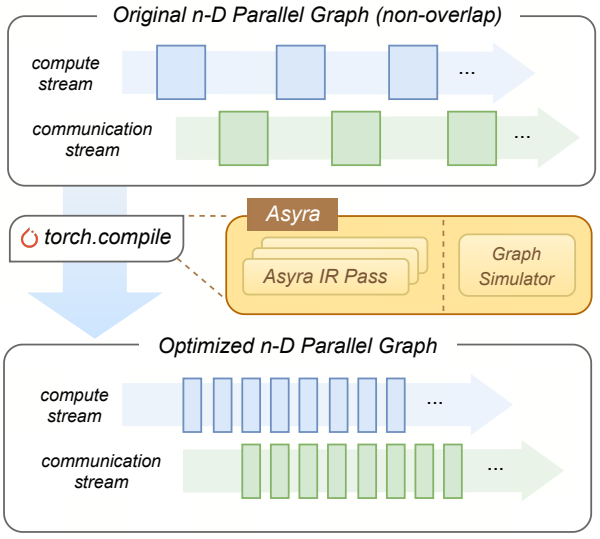


Figure 1. Overview of ASYRA workflow. The execution graph of n-D parallelism is optimized by ASYRA compilation passes to improve communication–computation overlap, thereby reducing communication bubbles and activation memory usage.

1. Introduction

Large models (LMs) have driven remarkable progress across language, vision, speech, and multimodal tasks, with transformer-based architectures scaling to billions of parameters (Brown et al., 2020; GPT, 2024). Training and inference of these models demand massive memory and computational resources, typically via model parallelism in high-end GPU clusters (Rasley et al., 2020; Korthikanti et al., 2023). As model scales and context lengths continue to grow, efficient large-scale distributed training and inference are of vital importance.

One prevalent challenge in optimizing large-scale parallelism is to minimize the overhead of communication bubbles induced by model parallelism, which can dramatically reduce overall hardware utilization. For example, expert parallelism (EP) for Mixture-of-Experts (MoE) (Shazeer et al., 2017) architecture introduces communication bubbles due to the dynamic routing of tokens across distributed experts, necessitating All-to-All communication primitives. Recent frameworks (Luo et al., 2025; Jin et al., 2025) report that communication bubbles induced by EP usually occupy up to 40% of the walltime for training. Consequently, effectively overlapping communication with computation for the complex multi-dimensional parallelism becomes a critical optimization challenge.

While prior works have explored the overlap, they typically struggle to balance performance with generality. Kernel-centric approaches, such as Ring-Attention (Liu et al., 2023) for sequence parallelism and Flux (Chang et al., 2024a) for tensor parallelism, achieve high overlap efficiency. However, they rely on bespoke, labor-intensive kernel implementations tailored to specific computation-communication patterns. Similarly, while Triton-distributed (Zheng et al., 2025a) exposes communication capabilities to Triton ker-

[*]Equal contribution [1]Unaffiliated. Correspondence to: Zhisheng Ye <yezhisheng@pku.edu.cn>, Lei Zhang <leizhang.real@gmail.com>.

Preprint. January 30, 2026.

nels, it still necessitates complex manual hand-programming to orchestrate the overlap. Conversely, compiler-based methods like SimpleFSDP (Zhang et al., 2024) attempt to automate this process by reordering intermediate representations (IRs). However, these solutions are often tethered to specific parallel patterns (e.g., data parallelism). Consequently, these solutions suffer from limited flexibility, requiring significant manual re-engineering whenever model architectures or parallel strategies change, failing to provide a unified automatic solution for complex n-D parallelism.

More fundamentally, enabling automatic communication-computation overlap in n-D parallelism is challenging due to barriers in abstraction, correctness, and optimality. First, the lack of a unified abstraction ties existing approaches to specific parallel or computation patterns. Second, fine-grained interleaving complicates data-dependency correctness, where manual synchronization is error-prone and hard to debug. Finally, prior works generally focus on local, intra-pattern optimizations, thereby ignoring global scheduling opportunities that span across different parallel dimensions (e.g., overlapping operations between SP and TP stages), preventing them from achieving optimality. We elaborate on them in §2.3.

To address these challenges, we propose ASYRA, an automated graph scheduling framework that unifies n-D parallel optimization within the compiler. ASYRA operates through three sequential compile passes (§3.3) after decomposing the execution graph into a unified torch.compile IRs. First, a *Tiling Pass* decomposes operators into independent tile streams. This transformation converts complex synchronization requirements into explicit graph dependencies and guarantees correctness. Second, a *Reordering Pass* tackles global scheduling by interleaving these streams, strategically advancing communication operators into the computation bubble of independent patterns across the entire graph. Finally, a *Bucketing Pass* mitigates the potential overhead of tiling by selectively fusing kernels based on runtime cost estimates to maximize efficiency.

The main contributions of this work are as follows:

- We propose ASYRA, an automated compiler-based framework designed to unify communication-computation overlap for general n-D model parallelism. ASYRA operates on a unified graph abstraction, enabling scalable and flexible optimization across diverse parallel configurations.
- We formulate the overlap problem as constrained graph scheduling and present a novel three-pass optimization pipeline guided by a lightweight graph simulator.
- We conduct extensive evaluations on popular LLMs, demonstrating ASYRA's superior efficiency on throughput and memory usage, consistently outperforming both baselines and existing overlap techniques.

2. Background and Challenge

2.1. Model Parallelism

Model parallelism is essential for training and inference of billion-parameter LMs that exceed single-device memory and compute limits. Early efforts (Narayanan et al., 2021; Rasley et al., 2020; Li et al., 2023) introduced tensor parallelism (TP) and pipeline parallelism (PP) to distribute intra-layer tensors and layer groups across devices individually, enabling billion-parameter Transformers training across thousands of GPUs. Subsequent advances extended parallelism beyond dense Transformers, including expert parallelism (EP) for Mixture-of-Experts models (He et al., 2021; Rajbhandari et al., 2022; Hwang et al., 2023) and sequence parallelism (SP) to reduce activation memory for long-context training (Jacobs et al., 2023; Liu et al., 2023). In practice, contemporary parallel frameworks (Liang et al., 2024; Li et al., 2025) compose these strategies across multiple dimensions, giving rise to n-D parallelism with increased scalability and scheduling complexity. When communication overhead induced by parallelism is exposed on the critical path, it directly translates into execution bubbles, leaving GPU compute resources idle.

2.2. Communication-Computation Overlap

To mitigate the computational bubble bottlenecks of n-D parallelism, the dominant research adopts communication-computation overlap. Rather than waiting for communication to complete, these approaches schedule communication concurrently with other computations whose execution does not depend on the communicated data, thereby hiding communication latency and reducing device idle time. Existing methods can be broadly classified into two categories.

Kernel-centric approaches exploit fine-grained asynchrony, interleaving collective communication with local computation within the kernel. Ring-Attention (Liu et al., 2023) partitions long sequences in SP into blocks and enables the communication of remote blocks to be fully overlapped with local attention computation. Flux (Chang et al., 2024b) targets TP workloads by over-decomposing large operations and fusing computation with communication into unified GPU kernels. AsyncTP (Liang et al., 2024) and FlashOverlap (Hong et al., 2025) further exploits fine-grained pipelining by splitting TP collectives into smaller chunks and overlapping them with partial matrix multiplications. Triton-Distributed (Zheng et al., 2025a;b) extends the Triton kernels with distributed primitives (e.g., OpenSHMEM), enabling hand-programming asynchronous communications to be scheduled with computation overlapping.

Compiler-based methods exploit graph-level analysis and static scheduling to expose communication-computation overlap between operators. CoCoNet (Jangda et al., 2022)

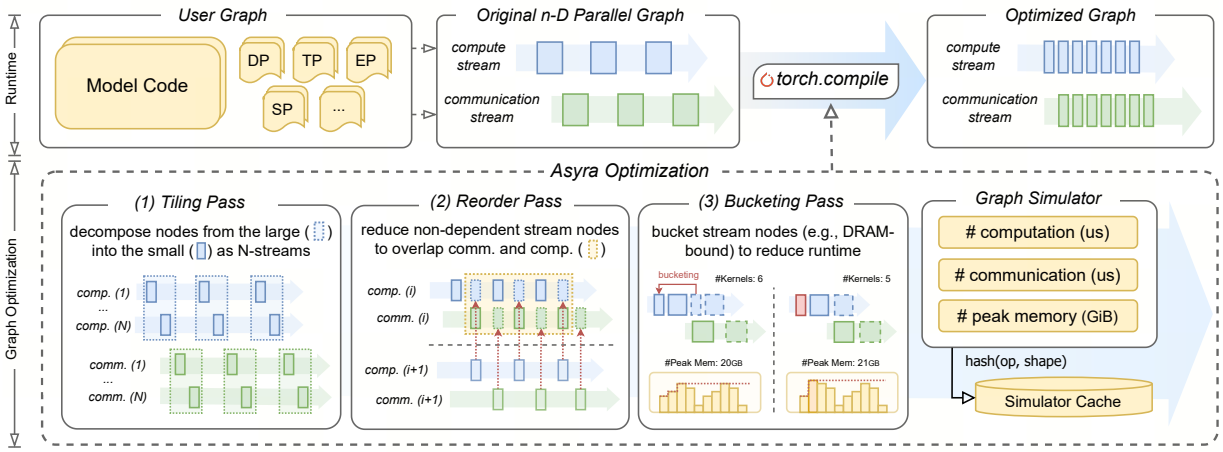


Figure 2. Workflow of ASYRA optimization. The n-D parallel graph is optimized during the ASYRA compilation stage using three passes: (1) The tiling pass decomposes the graph into data-independent K streams; (2) The reorder pass tries to overlap communication and computation across K streams; (3) The bucketing pass fuses fragmented IRs to reduce overhead without breaking the overlap. In addition, the Graph Simulator estimates runtime and peak memory usage to guide the optimization passes above.

introduces a compiler that breaks the rigid synchronization between computation and communication by reordering operator execution, enabling fine-grained overlap between matrix multiplication and communication. Lancet (Jiang et al., 2024) targets EP workloads by modeling MoE execution as a computation graph and scheduling operators around All-to-All communication to maximize overlap with independent computation. SimpleFSDP (Zhang et al., 2024) leverages compiler-captured training FSDP graphs to reorder and prefetch sharded parameter communication, overlapping communication with subsequent computation. Pfeife (Jhoo et al., 2025) and JaxPP (Xhebraj et al., 2025) introduce the compile-based automatic pipeline parallel scheduling approach in TorchDynamo and JAX compiler individually.

2.3. Challenges towards Automated n-D Parallel Overlap

Prior works suffer from limited flexibility, requiring either manual tuning or being tethered to a specific parallelization strategy. They can also become less effective and require significant manual effort to develop and tune when hardware, model architectures, or parallelism strategies change. More fundamentally, enabling *general-purpose* automated communication–computation overlap in n-D parallelism faces three distinct challenges:

(1) Lack of a Unified Abstraction. Existing approaches are typically tethered to specific parallel patterns or fixed operator templates, rendering overlap strategies brittle to configuration changes. Overcoming this requires a compiler-level abstraction capable of decoupling optimization policies from specific model architectures. *Solution:* We address this by decomposing n-D parallel execution into primitive computation and communication operators within the unified graph representation induced by `torch.compile`.

(2) Complexity of Preserving Data-Dependency Correctness. Fine-grained overlap traditionally relies on explicit, manual synchronization mechanisms, which are error-prone and difficult to scale across complex graphs. A general solution must instead rely on automated mechanisms to inherently preserve execution order without intrusive, ad-hoc barriers. *Solution:* We enforce correctness by applying graph tiling and other graph transformations while strictly respecting the original topological data dependencies, thereby avoiding the need for manual synchronization.

(3) Missed Opportunities without Global Scheduling. Overlap opportunities are inherently global and often span multiple parallel dimensions. However, prior works are largely restricted to local or pattern-specific optimizations. *Solution:* We perform scheduling across the whole graph, systematically exposing and exploiting cross-dimensional overlap opportunities that are invisible to local optimization scopes.

3. Methodology

3.1. Problem Formulation

Overlapping communication and computation in model parallelism can be formulated as the model-graph scheduling via graph transformations. Let $G = (V, E)$ denote a model graph equipped with n-D parallelism, where each node $v \in V$ is a compute or communicate operator, and each edge $(u \to v) \in E$ represents a precedence constraint that v execution starts after u finishes. To overlap graph communications and computations, we aim to find a transformation sequence $\pi = (\tau_1, \ldots, \tau_L)$ with variable length L to rewrite graph G into an optimized graph $G^* = f_\pi(G)$ that exposes more communication-computation overlap opportunities while preserving execution correctness. Specifically, we

consider the following graph transformations: (1) **Tiling**, denoted as $\tau_{\text{tiling}} : v \mapsto \{v_1, \ldots, v_K\}, K \geq 2$, can partition a long communicate or compute operator v into K-tiles $(v_1 \to \cdots \to v_K)$ to reveal an overlap opportunity, such as hidden computation tiles under the communications. (2) **Reordering**, denoted as $\tau_{\text{reorder}} : E \mapsto E'$, aims to adjust the topological order among independent nodes without violating dependencies, e.g., starts communication operator earlier or delays its corresponding wait operator to enlarge the computation overlap spans; (3) **Bucketing**, denoted as $\tau_{\text{bucketing}} : \{v_1, \ldots, v_p\} \mapsto v,\ p \geq 2.$, can merge a set of homogeneous operators $\{v_1, \ldots, v_p\}$ into a single node v to improving memory and computation efficiency, reducing the potential overhead of tiling. Through these transformations, the optimized graph $f_\pi(G)$ is expected to exhibit an improved communication-computation overlap. We evaluate this improvement using the makespan $T(\cdot)$ which denotes an execution-time simulator of the graph. Formally, we present an objective in Eq. (1),

$$\begin{aligned} &\arg\min_{\pi} T(f_\pi(G)) \\ &\text{s.t.} \quad \texttt{PeakMem}(f_\pi(G)) \leq M_{\text{budget}} \end{aligned} \tag{1}$$

where for a given model graph G, we aim to minimize the makespan of the communication-computation overlapped graph $f_\pi(G)$ by transformation π, while ensuring that the peak memory consumption $\texttt{PeakMem}(f_\pi(G))$ remains below a specified budget M_{budget}, typically corresponding to the available per-GPU memory capacity.

In practice, directly solving this constrained optimization is intractable. ASYRA therefore adopts a sequence of heuristic, graph-guided optimization passes that progressively improve overlap under memory constraints.

3.2. System Overview

As illustrated in Fig. 2, ASYRA is designed as a graph transformation backend integrated into the PyTorch compilation stack. It operates purely at the intermediate representations (IR) level, taking the execution graph captured by `torch.compile` as input and producing an optimized schedule for communication-computation overlap.

The optimization passes (§3.3) proceed sequentially, guided by a lightweight Graph Simulator (§3.4). First, the tiling pass decomposes the rigid operator graph into flexible, independent tile streams, forming a unified abstraction across different parallelism. Next, the reordering pass performs global scheduling, interleaving these streams to overlap execution bubbles while respecting data dependencies. Finally, the bucketing pass further improves communication and computation efficiency by mitigating the potential overhead introduced by tiling, using the simulator's cost model to selectively fuse kernels. We denote an illustrative example of the efficacy of the three passes in Fig. 3. This design decouples optimization from specific model architectures and parallel strategies, making ASYRA composable with existing parallel strategies and other overlap strategies.

Unified Composability. By operating on the unified IRs, ASYRA inherently supports diverse parallel strategies (e.g., TP, EP, SP) and compatible overlap techniques (e.g., Async-TP) without requiring strategy-specific adaptations. Since these methods manifest as standard compute and communication nodes within the compiled SPMD-style graph, ASYRA schedules them uniformly, enabling joint optimization across mixed parallel dimensions. Note that this relies on the visibility of the graph to compilers. Techniques represented as explicit IRs can be fully optimized, while opaque custom operators default to conservative scheduling to guarantee correctness for ASYRA, unless additional annotations are provided.

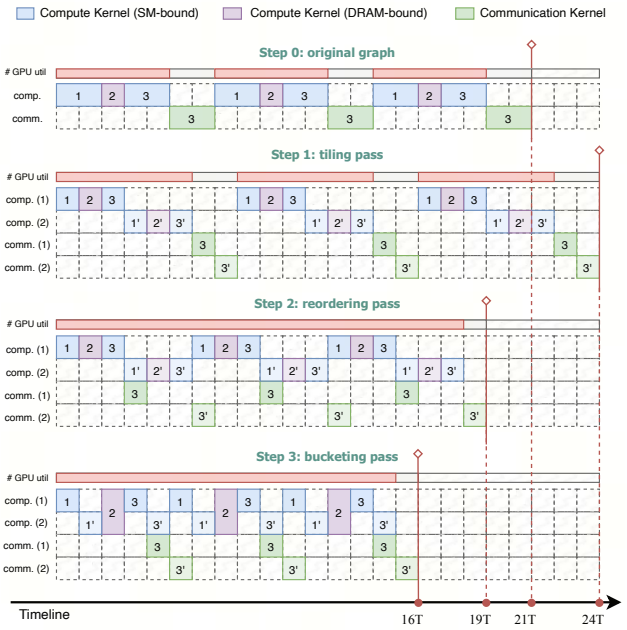


Figure 3. Illustration of the efficacy of Tiling, Reordering, and Bucketing in ASYRA. Tiling (Step 1) initially chunks the graph to create overlapping opportunities. Then Reordering (Step 2) reorders node 3, interleaving computation and communication to minimize bubbles. Finally, Bucketing (Step 3) reduces the makespan to 16T by bucketing kernels (node 2).

3.3. Optimization Pass

In this section, we introduce three major ASYRA compiler optimization passes, which are applied sequentially to transform the compiled model graph with communication-computation overlap.

3.3.1. GRAPH TILING

Graph tiling can partition the compiled execution graph into multiple independent tile streams, where each stream is a se-

quence of tiles derived from the original operators. The motivation of graph tiling is to break the coarse-grained dependency coupling between communications and computations in the original graph, and reconstruct multiple independent streams mutually, so that communication tiles in one stream can have the opportunity to be interleaved and overlapped with compute tiles in other streams. Formally, given a model graph $G = (V, E)$, the tiling pass constructs K subgraphs $\{G_i = (V_i, E_i)\}_{i=1}^{K}$, where $V_i = \{\, v_i \in \tau_{\text{tiling}}(v) \mid v \in V \,\}$ denotes the set of the i-th tiles of all original operators, and $E_i = \{(u_i \to v_i) \mid u_i, v_i \in V_i\}$ preserves the data-dependency structure within each stream. ASYRA applies the tiling function $\tau_{\text{tiling}}(\cdot)$ along batch-size dimension by default for best generality and preserve correctness, and optionally supports configuring the tiling dimension (e.g., sequence dimension) for correctness-guaranteed regions.

Benefits and Potential Overheads. The tiling pass significantly alleviates peak memory pressure. By decomposing large-scale operators into finer-grained tiles, it reduces the size of inputs and intermediate activations that must be materialized simultaneously, effectively lowering the peak memory during execution. However, tiling would degrade the computation and communication performance due to (i) reduced kernel efficiency for memory-bound operators (e.g., layer normalization, softmax, and point-wise operations), (ii) fragmented communication that would lower effective bandwidth and throughput, (iii) increased kernel launch and scheduling. Performance profiling in Appendix §A.1.1 shows that performance typically degrades for these operators as the tiling factor increases. We empirically find that a small tiling factor, e.g., $K = 2$, usually provides a reasonable trade-off between exposing overlap and limiting fragmentation. Additionally, we introduce the bucketing pass (§3.3.3) to further mitigate its effects.

Algorithm 1 Segmented Round-Robin Reordering (SegRR)

Input: K-tile streams IR orders $\{\mathcal{O}_i\}_{i=1}^{K}$
Output: global overlap scheduled IR order $\mathcal{O}_g$

Initialize global overlap scheduled order $\mathcal{O}_g \leftarrow \emptyset$
for *tile* $i = 1, 2, ..., K$ **do**
  Initialize segments $\mathcal{S}_i \leftarrow \{\}$
while *exists non-empty* $\mathcal{O}_i$ **do**
  for *tile stream* $i = 1, 2, ..., K$ **do**
    if $\mathcal{O}_i$ *is empty* **then**
      continue
    $v \leftarrow \texttt{pop}(\mathcal{O}_i)$
    /* Gathering IR segments */
    if $\texttt{type}(v) \in \{\texttt{compute}, \texttt{communicate}\}$ **then**
      $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{v\}$
    /* Overlapping other IR segments if collective wait reaching */
    if $\texttt{type}(v) \in \{\texttt{wait}\}$ **then**
      while $\mathcal{S}_i \neq \emptyset$ **do**
        $u \leftarrow \texttt{pop}(\mathcal{S}_i)$
        $\mathcal{O}_g \leftarrow \mathcal{O}_g \cup \{u\}$
      $\mathcal{S}_i \leftarrow \{v\}$
    /* Handling outlier operators */
    if $\texttt{type}(v) \in \{\texttt{other}\}$ **then**
      $\mathcal{O}_g \leftarrow \mathcal{O}_g \cup \{v\}$
/* Handling boundary IR segments */
for *tile segment* $i = 1, 2, ..., K$ **do**
  while $\mathcal{S}_i \neq \emptyset$ **do**
    $u \leftarrow \texttt{pop}(\mathcal{S}_i)$
    $\mathcal{O}_g \leftarrow \mathcal{O}_g \cup \{u\}$

3.3.2. SEGMENTED REORDERING

After the tiling pass, the graph is split into multiple independent tile streams. The reordering pass need to adjust the launch order across operators, that maximizes communication-computation overlap across these streams while preserving data-dependency. We propose Segmented Round-Robin Reordering (**SegRR**), a segment-wise round-robin manner to overlap communications and computations while controlling the peak memory under memory budgets. Specifically, SegRR is designed with two objectives: **(i) practically improving overlap** by advancing communication launches into the bubble of computation on other streams, and **(ii) controlling in-flight memory** by avoiding overly fine-grained interleaving that keeps many partially-executed tiles in flight simultaneously.

Instead of strictly serializing operations or naively interleaving them, SegRR adopts a segment-wise scheduling strategy to maximize overlap while respecting synchronization constraints. The core insight is to treat explicit synchronization primitives (i.e., `wait` operators) as natural boundaries that divide a continuous stream into independent segments. As illustrated in Algorithm 1, within each round, SegRR identifies and schedules the maximal contiguous block of non-blocking operators (`computation` and `communicate` operators) from a stream until a dependency barrier is reached. The formed segment is scheduled as a whole part. Therefore, it allows communication tasks from one stream to be effectively injected into the computation bubbles of others, pipelining execution across parallel dimensions without violating data correctness.

Crucially, the segment-wise scheduling in SegRR enforces memory-bounded execution, compared with operator-wise scheduling. Unlike aggressive scheduling that might launch massive amounts of asynchronous operators eagerly, SegRR flushes pending segments before processing synchronization barriers. This ensures that the intermediate tensors (activa-

tions and buffers) associated with a segment are consumed and released promptly upon the segment's completion. By limiting the number of in-flight segments active simultaneously, SegRR prevents the peak memory explosion, effectively navigating the trade-off between overlap efficiency and memory footprint.

3.3.3. COST-AWARE BUCKETING

While the tiling pass effectively exposes fine-grained overlap opportunities, it may introduce potential overhead for specific operations, as discussed in §3.3.1. The bucketing pass attempts to selectively bucket identical operators across different tile streams back into single kernels while maintaining a high overlapping opportunity.

There are two trade-offs the bucketing pass needs to consider. First, merging independent tiles from different streams into a single bucket reintroduces synchronization barriers and reduces the flexibility to interleave these operators with communication, potentially negating the overlap gains from tiling. Second, to form a bucket, operators originally scheduled for later execution must be advanced. This premature execution extends the lifetime of intermediate tensors, thereby increasing peak memory usage.

To navigate these trade-offs, ASYRA formulates bucketing as a constrained search problem. Instead of blindly merging all available tiles, we formulate a search-based algorithm that evaluates candidate groups of mergeable operators. For a candidate bucket formed by merging a subset of tiles $\{v_1, \ldots, v_K\}$, the bucketing pass applies the transformation $\tau_{\text{bucketing}}(\cdot)$ to merge these tiles together and topological resort the dependent prior operators to obtain a tentative graph $\tilde{G}$. The graph simulator is then invoked to estimate its makespan $T(\tilde{G})$ and peak memory $\text{PeakMem}(\tilde{G})$. By considering the extra peak memory as costs and the makespan saving as values for each possible merge, we apply dynamic programming to efficiently find the optimal decision. This ensures that ASYRA recovers hardware efficiency without violating memory constraints or destroying the critical communication overlap established by prior passes.

3.4. Graph Simulator

ASYRA employs a lightweight graph simulator to guide the above graph transformations, by providing estimations of execution time and peak memory consumption under different scheduling choices. The simulator operates purely at compile time and relies only on graph-visible information without heavy real runs, making it deterministic, lightweight, and composable with existing parallel strategies.

The graph simulator estimates computation and communication cost by integrating analytical modeling with automated benchmarking, using operator semantics and tensor shapes alongside synchronized real-world measurements to ensure robust computation and communication cost profiles. Furthermore, by simulating the execution order and tracking the lifetimes of intermediate tensors and buffers, the simulator provides a precise, conservative estimate of peak memory consumption.

4. Evaluation

4.1. Experimental Setting

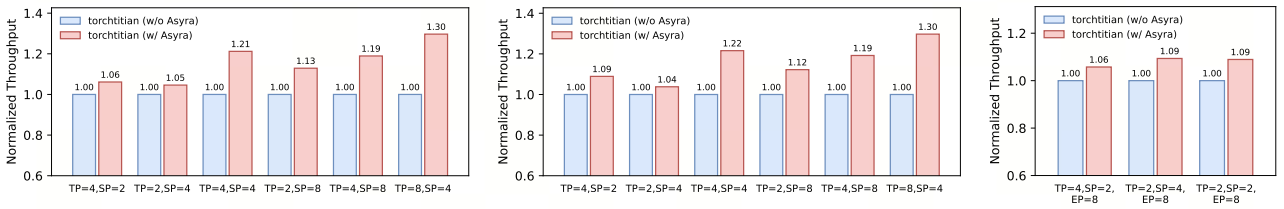
We evaluate ASYRA on popular LLMs, including GPT-3, LLaMA3, and Qwen3-MoE. All experiments are conducted on NVIDIA A800 SXM4 GPUs with 400GB/s intra-node NVLink and 4×200 Gbps inter-node RoCE links. Our implementation builds upon TorchTitan (Liang et al., 2024), which also serves as the baseline. We integrate ASYRA directly into it, leveraging TorchTitan's model-parallel capabilities[1]. The software stack also includes the official PyTorch 2.9 and its corresponding Docker images. We conduct a comprehensive evaluation of ASYRA, encompassing its performance and peak memory efficiency, and detailed ablation studies of each system component.

4.2. Efficiency of ASYRA

As shown in Fig. 4, we evaluate the end-to-end training throughput of ASYRA. Compared to the baseline, ASYRA yields consistent gains in both settings: $4\% \sim 9\%$ within intra-node model parallelism and $12\% \sim 30\%$ when scaling model parallel across multiple nodes. Regarding system overhead, ASYRA incurs a negligible, one-time compilation cost of less than 2 minutes to execute the three optimization passes before each experiment.

To better understand where the speedups come from, we further examine ASYRA's efficiency in detail. Specifically, Table 2 reports per-step training latency, communication bubble, and model FLOPs utilization (MFU), broken down by layer-wise forward and backward. Under intra-node model parallelism, ASYRA reduces the forward and backward communication bubbles by more than 60%, accounting for less than 8% bubble time remaining in each layer. Under inter-node model parallelism, ASYRA achieves larger bubble reductions, as cross-node bandwidth limitations are more likely to be dominant. For example, in 128K long-context training of LLaMA3-8B with TP=4 and SP=8, ASYRA reduces the communication bubble by nearly 50% and increases MFU by 9% and 5% for the forward and backward steps, respectively. These results indicate that ASYRA improves training throughput by enabling more effective communication-computation overlap.

[1] In our experiments, tensor parallelism (TP) is built on PyTorch DTensor, sequence parallelism (SP) follows the Ulysses strategy, and expert parallelism (EP) is implemented as TorchTitan EP style.

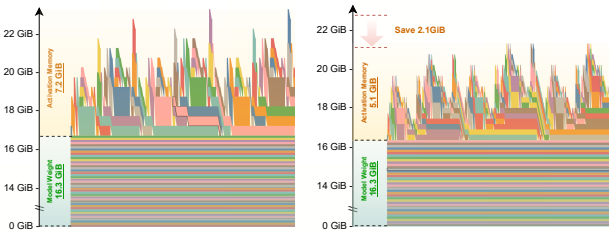


(a) GPT3 Curie (6.7B) (b) LLaMA3 (8B) (c) Qwen3-MoE (30B-A3B)

Figure 4. End-to-end training throughput speedup of ASYRA over TorchTitan baseline under n-D model parallelism (TP/SP/EP) on three large language models, achieving up to $4\% \sim 9\%$ gain for intra-node parallelism and $12\% \sim 30\%$ speedup for inter-node parallelism.

4.3. Peak Memory Saving

As illustrated in Fig. 5, we profile the allocated-memory timeline during prefill inference with ASYRA on LLaMA3-8B at a 64K sequence length, using sequence parallelism (SP=4). Compared with the baseline (w/o ASYRA), our approach (w/ ASYRA) effectively reduces activation memory by up to 30%, and lowers the peak memory footprint from 23.5 GiB to 21.4 GiB. The evident peak-memory reduction mainly comes from both the tiling pass and the memory-efficient reordering pass (SegRR). SegRR improves overlap while staying memory-friendly because it groups work into short segments and ends each segment with an explicit wait, which acts as a clear lifetime boundary and eliminates improper lifetime extension of intermediate tensors, compared to normal operator-wise scheduling.



(a) w/o ASYRA (Baseline) (b) w/ ASYRA (Ours)

Figure 5. The comparison of allocated memory of LLaMA3 (8B) on the inference stage, where ASYRA can save up to 30% activation memory compared with the baseline (w/o ASYRA).

4.4. Efficiency over Other Overlap Technique

As shown in Fig. 6, we compare the training performance of ASYRA with AsyncTP (Liang et al., 2024), which leverages hand-tuned kernels to fuse computation and communication. Results show that ASYRA consistently outperforms AsyncTP across a range of large models and parallel configurations. Specifically, on GPT3-Davinci 175B training, ASYRA achieves a 17% speedup over the baseline, surpassing AsyncTP's 12%. Similarly, for LLaMA3-70B, ASYRA achieves a 16% speedup, compared to their 13%.

This advantage stems from two key factors. First, ASYRA achieves overlap with fewer computation kernels compared to AsyncTP due to bucketing, thereby avoiding the overhead of frequent kernel launches. Second, ASYRA is free of synchronization due to the explicit graph dependencies, whereas AsyncTP utilizes symmetric memory (Wang et al., 2025) for synchronization, diminishing the effectiveness of asynchronous execution. A more comprehensive analysis is provided in Appendix §A.2. In general, these structural advantages translate directly into our findings, where ASYRA surpasses the hand-tuned AsyncTP in overall performance.

Table 1. End-to-end comparison of runtime and peak memory between real trace and graph simulator.

	Runtime (sec)		PeakMem (GiB)	
Model Configuration	**EST.**	**REAL**	**EST.**	**REAL**
GPT3 Curie (TP=2, SP=4)	6.80	6.41	65.9	66.0
↪ w/ ASYRA overlap	6.61	6.12	65.4	65.7
LLaMA3 8B (TP=2, SP=4)	7.11	6.95	69.7	70.1
↪ w/ ASYRA overlap	6.95	6.73	69.5	69.8
Qwen3 30B-A3B (TP=2, SP=2, EP=8)	8.72	8.63	72.3	74.5
↪ w/ ASYRA overlap	8.23	8.15	72.3	74.8

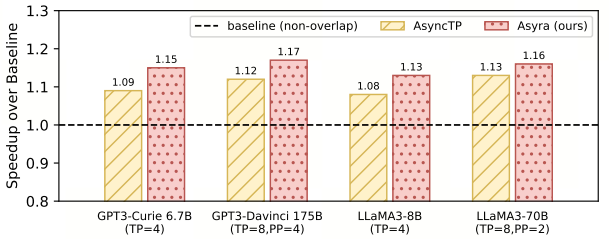


Figure 6. The comparison of the per-step training speedup between ASYRA and other overlap approaches (AsyncTP) with TP enabled.

4.5. Fidelity of Graph Simulator

We validate the fidelity of ASYRA's graph simulator by comparing its estimates with real execution traces on end-to-end per-step training runtime and peak memory.

As shown in Table 1, the simulator provides accurate predictions for both metrics across different model-parallel configurations. For example, on the GPT3 Curie (TP=2 and SP=4), the estimated training runtime latency is 6.80 seconds, closely matching the real runtime of 6.41 seconds, whereas the estimated peak memory of 65.9 GiB is almost

Table 2. The comparison of layer-wise training performance under various parallel settings on three different LLMs with ASYRA.
(Intra-node: TP × SP × EP $\leq$ 8; Inter-node: TP × SP × EP $>$ 8)

TP	SP	EP	ASYRA	Forward				Backward			
				LATENCY ↓ (ms)	BUBBLE ↓ (%)	TFLOPS ↑ (half)	MFU ↑ (%)	LATENCY ↓ (ms)	BUBBLE ↓ (%)	TFLOPS ↑ (half)	MFU ↑ (%)
GPT3 Curie (6.7B)											
4	2	/	×	64.6	21.2	161.7	51.8	127.6	10.2	163.6	52.4
			✓	58.4	7.7	178.6	57.2	122.6	3.3	170.3	54.6
2	8	/	×	331.1	31.2	126.2	40.4	665.4	15.1	125.6	40.2
			✓	275.9	16.3	151.4	48.5	606.7	6.4	137.7	44.2
4	8	/	×	735.8	44.8	99.2	31.8	1,355.2	24.2	107.7	34.5
			✓	572.9	26.2	127.4	40.8	1,185.1	12.3	123.1	39.4
LLaMA3 (8B)											
4	2	/	×	90.8	15.7	140.7	45.1	196.7	7.7	129.9	41.6
			✓	85.5	5.5	149.3	47.8	191.4	2.7	133.5	42.8
2	8	/	×	340.7	30.3	126.6	40.6	690.7	15.8	124.9	40.0
			✓	287.1	15.5	150.3	48.2	631.9	7.2	136.6	43.7
4	8	/	×	753.4	44.0	98.7	31.6	1,387.6	24.9	107.2	34.3
			✓	584.8	25.8	127.1	40.7	1,211.2	12.3	122.7	39.3
Qwen3-MoE (30B-A3B)											
2	2	8	×	61.7	31.2	126.7	40.6	115.4	15.2	135.6	43.5
			✓	53.2	15.3	147.1	47.1	109.4	7.8	142.9	45.8
2	4	8	×	84.7	22.9	127.9	40.9	178.7	11.0	121.2	38.8
			✓	77.6	11.5	139.5	44.7	171.5	4.6	126.3	40.5
4	2	8	×	121.2	31.8	89.3	28.6	227.1	15.8	95.3	30.5
			✓	105.8	16.3	102.3	32.8	212.8	8.3	101.8	32.6

identical to the real 66.0 GiB. Nevertheless, we consider that the remaining runtime gap mainly comes from (i) being agnostic to backend optimizations such as kernel fusion and kernel-level tuning, and (ii) runtime overheads and instabilities (e.g., kernel-launch overhead and machine-level variability) introduce additional variance. Appendix §A.1.2 provides a more comprehensive analysis. Overall, these negligible differences are acceptable for scheduling guidance.

Table 3. Effectiveness of ASYRA optimization passes on Qwen3-MoE decoding stage of each layer with EP=8.

Tiling (§3.3.1)	Reordering (§3.3.2)	Bucketing (§3.3.3)	Latency (ms)	Bubble (%)	Speedup
Baseline (w/o ASYRA)			3.36	44.6	/
✓			4.14	31.8	×0.81
✓	✓		3.35	**9.5**	×1.01
✓	✓	✓	**2.94**	13.2	×**1.14**

4.6. Ablation Study

To evaluate the individual and cumulative contributions of ASYRA optimization passes, we conduct an ablation study on Qwen3-MoE decoding inference with EP=8 to verify the contributions of each pass on latency, bubble, and speedup. As shown in Table 3, tiling alone incurs a performance penalty (down to $0.81\times$) due to fragmentation overhead, creating the necessary granularity and opportunities for overlapping execution. The addition of reordering mitigates this by further overlapping computation and communication between tiling streams, drastically reducing execution bubbles from 44.6% to 9.5% and keeping latency close to the baseline. The full optimization pipeline, incorporating bucketing, achieves the optimal latency of 2.94 ms ($1.14\times$ speedup), after merging and bucketing the fragmented kernels from tiling. Notably, although Bucketing slightly increases the bubble ratio compared to Reordering alone, it yields the highest end-to-end throughput by significantly enhancing computational efficiency. These results underscore the synergistic effect of ASYRA 's three optimization passes in achieving high efficiency.

5. Conclusion

We introduce ASYRA, an automated graph scheduling framework that systematically maximizes communication-computation overlap in n-D model parallelism through compiler-based tiling, reordering, and bucketing optimizations. ASYRA offers a highly efficient, scalable, and general-purpose solution for large model parallelism.

Impact Statement

This paper presents work whose goal is to advance the field of machine learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Gpt-4, 2024. URL https://openai.com/gpt-4.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.

Chang, L.-W., Bao, W., Hou, Q., Jiang, C., Zheng, N., Zhong, Y., Zhang, X., Song, Z., Yao, C., Jiang, Z., Lin, H., Jin, X., and Liu, X. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024a. URL https://arxiv.org/abs/2406.06858.

Chang, L.-W., Bao, W., Hou, Q., Jiang, C., Zheng, N., Zhong, Y., Zhang, X., Song, Z., Yao, C., Jiang, Z., et al. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024b.

He, J., Qiu, J., Zeng, A., Yang, Z., Zhai, J., and Tang, J. Fastmoe: A fast mixture-of-expert training system. *arXiv preprint arXiv:2103.13262*, 2021.

Hong, K., Li, X., Liu, M., Mao, Q., Wu, T., Huang, Z., Chen, L., Wang, Z., Zhang, Y., Zhu, Z., et al. Flashoverlap: A lightweight design for efficiently overlapping communication and computation. *arXiv preprint arXiv:2504.19519*, 2025.

Hwang, C., Cui, W., Xiong, Y., Yang, Z., Liu, Z., Hu, H., Wang, Z., Salas, R., Jose, J., Ram, P., et al. Tutel: Adaptive mixture-of-experts at scale. MLSys '23, 2023.

Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

Jangda, A., Huang, J., Liu, G., Sabet, A. H. N., Maleki, S., Miao, Y., Musuvathi, M., Mytkowicz, T., and Saarikivi, O. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 402–416, 2022.

Jhoo, H. Y., Hur, C.-K., and Lopes, N. P. Pfeife: Automatic pipeline parallelism for pytorch. In *Forty-second International Conference on Machine Learning*, 2025.

Jiang, C., Tian, Y., Jia, Z., Zheng, S., Wu, C., and Wang, Y. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping. *Proceedings of Machine Learning and Systems*, 6:74–86, 2024.

Jin, C., Jiang, Z., Bai, Z., Zhong, Z., Liu, J., Li, X., Zheng, N., Wang, X., Xie, C., Huang, Q., et al. Megascale-moe: Large-scale communication-efficient training of mixture-of-experts models in production. *arXiv preprint arXiv:2505.11432*, 2025.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. In *Proceedings of Machine Learning and Systems*, MLSys '23, 2023.

Li, S., Liu, H., Bian, Z., Fang, J., Huang, H., Liu, Y., Wang, B., and You, Y. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 766–775, 2023.

Li, Y., Wan, C., Lin, Z., Zhu, H., Yang, J., Song, Z., Di, X., Wu, J., Shu, H., Bao, W., Peng, Y., Lin, H., and Chang, L.-W. vescale: Consistent and efficient tensor programming with eager-mode spmd, 2025. URL https://arxiv.org/abs/2509.07003.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Luo, S., Li, P., Peng, J., Wang, H., Cheng, Y., Chen, T., et al. Occult: Optimizing collaborative communication across experts for accelerated parallel moe training and inference. *arXiv preprint arXiv:2505.13345*, 2025.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. 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*, SC '21, 2021.

Rajbhandari, S., Li, C., Yao, Z., Zhang, M., Aminabadi, R. Y., Awan, A. A., Rasley, J., and He, Y. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pp. 18332–18346. PMLR, 2022.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Wang, Y., He, H., and Wehrstedt, L. Pytorch symmetricmemory: Harnessing nvlink programmability with ease, 2025. URL https://dev-discuss.pytorch.org/t/pytorch-symmetricmemory-harnessing-nvlink-programmability-with-ease/2798/1.

Xhebraj, A., Lee, S., Chen, H., and Grover, V. Scaling deep learning training with mpmd pipeline parallelism. *Proceedings of Machine Learning and Systems*, 7, 2025.

Zhang, R., Liu, T., Feng, W., Gu, A., Purandare, S., Liang, W., and Massa, F. Simplefsdp: Simpler fully sharded data parallel with torch.compile, 2024. URL https://arxiv.org/abs/2411.00284.

Zheng, S., Bao, W., Hou, Q., Zheng, X., Fang, J., Huang, C., Li, T., Duanmu, H., Chen, R., Xu, R., Guo, Y., Zheng, N., Jiang, Z., Di, X., Wang, D., Ye, J., Lin, H., Chang, L.-W., Lu, L., Liang, Y., Zhai, J., and Liu, X. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025a. URL https://arxiv.org/abs/2504.19442.

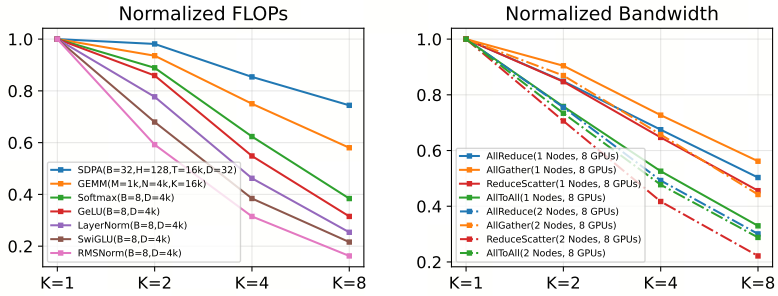
Zheng, S., Fang, J., Zheng, X., Hou, Q., Bao, W., Zheng, N., Jiang, Z., Wang, D., Ye, J., Lin, H., et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025b.

A. Appendix

A.1. Quantitative Study

A.1.1. Effectiveness of Tiling Factor

To investigate the effect of the tiling factor K on both computational throughput and communication efficiency, we benchmark a set of representative kernels on an NVIDIA A100 GPU with $K = 1, 2, 4, 8$. Fig. 7 shows that most kernels maintain acceptable performance at $K = 2$, whereas increasing K further (i.e., $K \geq 4$) results in a substantial degradation in both compute and bandwidth efficiency.



(a) Tiling impacts on computation (b) Tiling impacts on communication

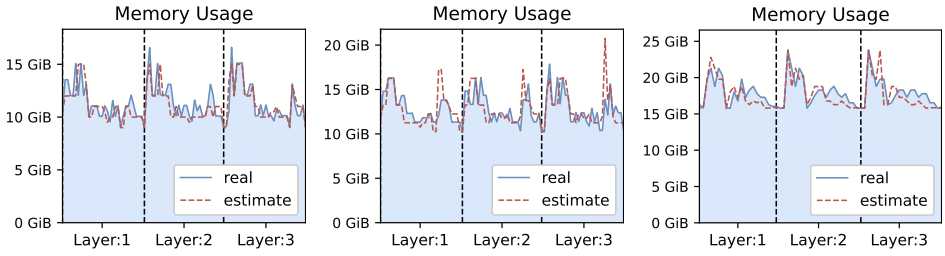
Figure 7. The computation and communication performance under different tiling factors ($K = 1, 2, 4, 8$).

- **Impacts on Computation.** Compute-bound operators (e.g., SDPA and GEMM) exhibit limited sensitivity to tiling at $K = 2$, with achieved FLOPs typically decreasing by no more than 10% with the tiling factor increasing. In contrast, memory-bound operators (e.g., Softmax and GeLU) are considerably more sensitive, where their achieved FLOPs drop by approximately 10%–20% with tiling factor K doubles.
- **Impacts on Communication.** Increasing the tiling factor tends to fragment communication into smaller messages, which reduces effective bandwidth due to higher per-message overhead. This effect is more noticeable for inter-node communication than for intra-node communication. When tiling factor $K = 2$, the intra-node bandwidth reduction is usually within 20%, while the inter-node bandwidth reduction can reach up to 30% in general.

Overall, these results suggest that $K = 2$ provides a favorable trade-off between increasing overlap opportunities and avoiding excessive fragment overhead, and we therefore adopt $K = 2$ as the default experimental setting.

A.1.2. Consistency of Memory Estimation

In Fig. 8, we compare the allocated-memory timeline from the real execution trace with that predicted by ASYRA graph simulator during the LLM prefill stage. The estimated memory closely matches the real trace, capturing not only the overall trend but also the peak occurrence with a similar timing profile. Overall, the simulator is sufficiently accurate to serve as a reliable prior for our optimization passes. In particular, it enables ASYRA to precisely profile candidate execution graphs and identify schedules that jointly control peak memory footprint and end-to-end makespan.

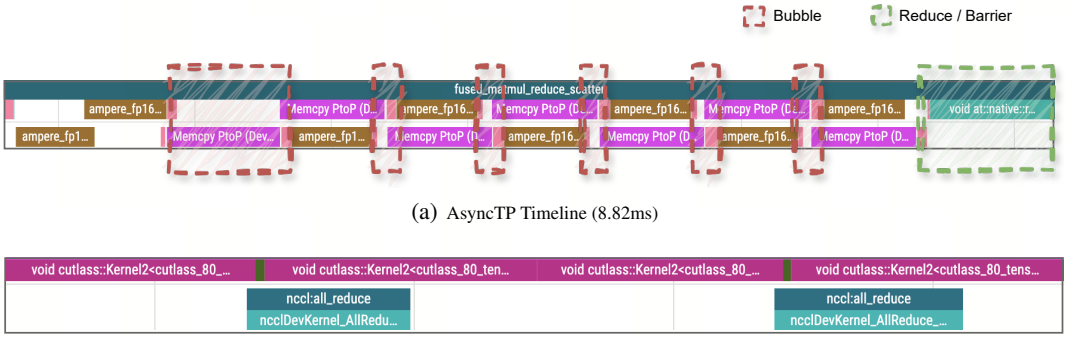


(a) GPT3 Curie *(TP=2, SP=2)* (b) LLaMA3 8B *(TP=2, SP=2)* (c) Qwen3 30B-A3B *(TP=2, SP=2, EP=4)*

Figure 8. The comparison of the allocated-memory timeline between real trace and graph simulator.

A.2. Timeline Visualization

In Fig. 9, we visualize the execution timelines of ASYRA and AsyncTP (Liang et al., 2024) on communication–computation overlap under tensor parallelism using `torch.profiler`. Compared to AsyncTP (step latency: 8.82 ms), ASYRA reduces the step latency to 7.67 ms, delivering a 13% speedup. We attribute this improvement to three factors.



(a) AsyncTP Timeline (8.82ms)

(b) ASYRA Timeline (7.67ms)

Figure 9. The Comparison of execution timeline of the communication–computation overlap between ASYRA and AsyncTP on tensor parallel. Blocks named with `nccl` / `memcpy` are communicate operators, and others are compute operators.

- **Less fragmentation overhead.** ASYRA partition computation and communication into only two tiles, which reduces fragmentation overhead (e.g., kernel launches and scheduling costs). In comparison, AsyncTP uses eight tiles (i.e., 4× finer granularity than ASYRA), which tends to exposing more fragmental costs than ASYRA.
- **Synchronization-free overlap.** ASYRA enables communication–computation overlap without introducing tile-level synchronization. This is because the ASYRA computation does not depend on its overlapped communication results. In contrast, AsyncTP must enforce synchronization to ensure that the overlapped computation results are ready before performing the next tile computation, which can introduce synchronization-induced bubbles and limit overlap.
- **Without post-communication reduction.** ASYRA performs the reduction *during* the communication phase. For example, in an `AllReduce` communication, the `sum` is finished as data is ring-exchanged across ranks. As a result, ASYRA avoids an additional post-communication reduction and/or barrier-like synchronization stage. By contrast, AsyncTP typically requires an extra synchronization at the end of the overlapped kernels.

Overall, ASYRA improves overlap efficiency by reducing fragmentation overhead, avoiding blocking synchronizations, and eliminating post-communication reduction, leading to a lower makespan in practice.